Approximate key and foreign key discovery in relational databases

by

Charlotte Vilarem

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Approximate key and foreign key discovery in relational databases

Charlotte Vilarem

Master of Science

Graduate Department of Computer Science

University of Toronto

2002

Organizations have been storing huge amounts of data for years, but over time, the knowledge about these legacy databases may be lost or corrupted.

In this thesis, we study the problem of discovering (approximate) keys and foreign keys satisfied in a relational database, of which we assume no previous knowledge.

The main ideas underlying our method for tackling this problem are not new; however, we propose two novel optimizations. First, we extract only key-based unary inclusion dependencies (UINDs) instead of all UINDs. Second, we add a pruning pass to the foreign key discovery; performed on a small data sample, it reduces the number of candidates tested against the database.

We validate our contributions on both real-life and synthetic data. We investigate the influence of input parameters on the performance of the pruning pass, and provide guidance on how to best set these parameters.

Acknowledgements

I would like to express my gratitude to a number of people without whose advice and encouragement this thesis would not have been possible.

First, I would like to thank my supervisor, Professor Renée Miller, for her guidance, advice, and patience during my research. I would also like to thank my second reader, Professor Ken Sevcik, for his time and helpful comments.

I would like to thank all my friends in Toronto, especially Travis, Vlad and Tristan, for the great time I had with you throughout my program at University of Toronto.

Finally, my deepest gratitude goes to my family and boyfriend for their support and encouragement, and for their involvement in this work. I would never have made it without them.

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

For more than a decade now, organizations have been storing huge amounts of data in relational databases. During the design phase of the database, a conceptual representation of the data is established (for instance an Entity-Relationship model), which describes the organization of the data in the database. This representation allows one to understand a dataset, and to develop applications using the data.

More precisely, a conceptual representation is used to explain the data, data relationships, data semantics, and data constraints. As an example, consider a university database. Its representation might specify that students are represented by a student number, a name and a firstname; courses are represented by a course number and a course title; and that students take several courses for their program, i.e., there is a many-to-many relationship between students and courses. It might also associate attribute names and their meaning, for instance: SNUM stands for student number.

In the relational data model, data are stored in flat tables, and each row represents an object or relationship in the real world. But the mere definition of tables is not enough to represent accurately the reality, we need to add *integrity constraints*

to ensure that the data stored in the database reflects accurately the real-world restrictions: for instance, that a student number is associated with only one student.

The most important integrity constraints, also called data dependencies, are functional and inclusion dependencies [CFP84]. Functional dependencies are conditions stating that for any value of a set of attributes, there is at most one value for a set of target attributes; for instance, given a room, a date and a time, there is at most one talk being held in there. Inclusion dependencies state that values occuring in certain columns of one relation must also occur in columns of another relation; for example, a manager is also an employee. In practice, most of the time, only specializations of these two types of dependencies are used: keys, and foreign keys. Keys identify uniquely objects (or tuples) in a table, e.g., a student number identifies uniquely a student. Foreign keys are inclusion dependencies that refer to a key; they express relationships between objects in the database. As an example, the column "student number" of a table "enrolled-in" is a foreign key of the key column "student number" in the table "student".

But over time, the conceptual representation may be lost, due to multiple updates, extensions, loss of documentation, or turnover of domain experts. Hence, these legacy databases may no longer be usable, and the knowledge they contain is wasted. Such a situation calls for a database reverse engineering process, in order to build a new model which reflects accurately the current state of the database. But because the data in legacy databases is likely to be noisy or corrupted, the true data semantics may not hold exactly in the database. Indeed, an exact model would incorporate the noise in its representation of the dataset, while what we really want is a model that reflects the data independent of the inconsistencies introduced with the time. Therefore, we may want to accomodate the noise with error thresholds, so that the

model found 'almost holds' in the database and fits the data without the noise.

However, determining dynamically the best error thresholds for a database is a difficult problem, so in this thesis, we focus on a simpler one: the discovery of approximate keys and foreign keys holding in a database instance for given approximation thresholds.

1.1 Motivation

Reverse engineering is the process of identifying a system's components and their relationships, and creating representations of the system in another form or at a higher level of abstraction [CI90]. Its aim is to redocument, convert, restructure, maintain, or extend legacy systems [Hai91]. In the case of database reverse engineering, this process can be divided into two distinct steps [PTBK96]: (1) eliciting the data semantics from the system, and (2) expressing the extracted semantics with a high level data model. Many approaches to database reverse engineering overlook the first step by assuming that some of the data semantics is known, but such assumptions may not hold for legacy databases. (For a classification of database reverse engineering methods according to their input requirements, see [dJS99].) Indeed, old versions of database management systems do not support the definition of foreign keys (e.g., foreign key definition was introduced in Oracle v7, released in 1992). Therefore, identifying the keys and foreign keys holding in a database instance can be part of the first step of a database reverse engineering process.

Integrity constraints, especially keys and foreign keys, can play an important role in query optimization. The role of query optimization is to find an efficient way

of processing a query, using all sorts of information about the data stored in the database, such as statistics and integrity constraints (keys and foreign keys). It is a three-stage process [Dat90]. First, the query optimizer rewrites the query into a more efficient form, so that the performance of the query does not depend on how the user chose to write it. Then, the optimizer generates possible execution plans for the new representation of the query. And then, it computes cost estimations for each of these execution plans and chooses the most efficient of them.

Keys and foreign keys are used both in the query rewrite phase, and to predict cost estimations. For example, if dept_number is a key of relation department, then the query "select distinct dept_num from department" is equivalent to the more efficient "select dept_num from department". Keys and foreign keys help in predicting the result size of joins, which is used in cost estimations.

Semantic query optimization, i.e., query optimization using a broader range of integrity constraints, has long been a subject of research, but most of the proposed techniques have not been implemented in commercial optimizers [CGK+99]. Cheng et al. and Godfrey et al. address this issue [CGK+99, GGXZ01] investigating how some semantic query optimizations using keys, foreign keys and other types of integrity constraints can be integrated in traditional query optimizers.

Integrity constraints can also be useful in schema mapping [MHH00]. The goal in schema mapping is to map a source (legacy) database into a fixed target schema through a set of queries on the source database. These mapping queries are derived from the correspondences between a target value and a set of source values. But when the value correspondences involve several relations of the source database, one needs a way of joining the tuples of these relations. The joins are determined from foreign key paths. So, these joins require knowledge of the integrity constraints (keys and

foreign keys) holding in the source database.

1.2 Contributions of the thesis

The contributions of this thesis are the following:

- We propose a method for (approximate) key and foreign key discovery in relational databases. Even though the underlying ideas are not new, this is the first proposal of a concrete algorithm for extracting foreign keys from a database of which we assume no previous knowledge.

- We implemented two novel optimizations for foreign key discovery. The more important is a pruning pass over the foreign key candidates, performed on a small data sample, which can reduce dramatically the number of accesses to the database. The other optimization consists of extracting solely key-based unary inclusion dependencies rather than all unary inclusion dependencies as is usually done, since only these are used in the foreign key candidate generation. This has a great impact in practice on the running time of the algorithm.

- We evaluate the performance of our algorithm on both publicly available real-life and synthetic datasets. This allows us to study the dataset characteristics that affect the algorithm.

1.3 Scope of the research

We provide in this thesis a general framework for the discovery of keys and foreign keys (FKs). We chose for simplicity to keep the data inside the DBMS and process it through SQL queries. But our architecture could easily be adapted to more efficient

treatments of the data.

The attributes whose types are large (such as long strings, e.g., varchar(3000)) were not considered for key or FK because they cannot appear in SQL statements of the form `SELECT DISTINCT ...` which are needed to test keys and FKs. In DB2, the limit is reached for data types larger than 255 bytes, and in Oracle, 4 Kbytes. This limitation should not be too restrictive though, because attributes with large types are generally not intended as keys or FKs. Indeed, large values in keys or FKs mean slower searches and useless replication of data, resulting in poor performance.

1.4 Outline of the thesis

In **Chapter 2**, we give some background and survey the literature about integrity constraints and their discovery.

In **Chapter 3**, we present our general architecture for approximate key and foreign key discovery, and analyze the complexity of our approach.

In **Chapter 4**, we describe the testing of our algorithm on both real-life and synthetic data, and study how input parameters and data characteristics influence the performance of our application.

Finally, we conclude in **Chapter 5**.

Chapter 2

Integrity constraints in relational databases

2.1 Background

2.1.1 Preliminaries

First of all, we define the relational database concepts used throughout this thesis. For more details, see the books by Levene and Loizou [LL99] or Date [Dat90].

Definition 2.1.1 (Basic relational database concepts). In the following, uppercase letters (which may be subscripted) from the end of the alphabet, such as X, Y, Z, will be used to denote sets of attributes, while those from the beginning of the alphabet, such as A, B, C will be used to denote single attributes.

A *relation schema* R is a finite set of attributes. The *domain* of an attribute A, denoted by $Dom(A)$, is the set of all possible values of A. A *tuple* over a relation schema $R = \{A_1, ..., A_m\}$ is a member of the Cartesian product $Dom(A_1) \times ... \times Dom(A_m)$. A *relation* r over R is a finite set of tuples over R. The cardinality of a

set X is denoted by $|X|$.

If $X \subseteq R$ is an attribute set, and t a tuple over R, we denote by $t[X]$ the restriction of t to X. The *projection* of a relation r over R onto X is defined by $\pi_X(r) = \{t[X] \mid t \in r\}$.

A *database schema* $\mathbf{R}$ is a finite set of relation schemas R_i. A *database* d over $\mathbf{R}$ is a set of relations r_i over each $R_i \in \mathbf{R}$.

Informally, relations are represented by tables, attributes by columns, and tuples by rows. The cardinality of a relation $|r|$ is the number of rows of the table; we will commonly denote it by p throughout this thesis.

We now define a running example used in the rest of this chapter.

Example 2.1.1. Consider the following database, called MovieDB, representing information about movies and the theater in which they are showing.

Table Movies

Title	Length	Genre
The mummy	90 min	Action
In the bedroom	90 min	Drama
Life of Brian	90 min	Comedy
Atanarjuat	180 min	Drama
Monster's ball	100 min	Drama

Table Theaters

Name	Location
ByTowne	Rideau Street
South Keys	Bank Street
Mayfair	Bank Street

Table Schedule

Theater	Movie	Time
ByTowne	In the bedroom	9pm
ByTowne	Monster's ball	4pm
South Keys	In the bedroom	3pm
South Keys	The mummy	8pm
South Keys	Monster's ball	3pm
Mayfair	Life of Brian	7pm

The database MovieDB is composed of three relations: Movies, with schema {Title, Length, Genre}, Theaters, with schema {Name, Location}, and Schedule, with schema {Theater, Movie, Time}. The database schema is the set {Movies, Theaters, Schedule}.

2.1.2 Functional and inclusion dependencies

The schemas alone are not sufficient to describe a database. Data semantics in a relational database are expressed by integrity constraints, and the most important of these constraints are functional and inclusion dependencies [CFP84, LL99]. Thus,

integrity constraints define the set of allowable states of a database.

Functional dependencies (FDs) have long been studied in the context of database normalization [LV00]. Database normalization is the process of designing a database satisfying a set of integrity constraints, efficiently and in order to avoid inconsistencies when manipulating the database [LL99]. In the presence of only functional dependencies, the problems that may arise are update anomalies (after an update, the data is no longer consistent with the functional dependencies) and redundency problems (parts of the data are unnecessarily duplicated); to prevent them, the database schema should be in Boyce-Codd Normal Form (BCNF) [Cod74]. A database schema is in BCNF if, in each relation of the database, all FD left-hand sides are keys or superset of keys of the relation. In the presence of both functional and inclusion dependencies (INDs), Levene and Vincent [LV00] identify additional problems and propose an inclusion dependency normal form (IDNF) to prevent them. A database is in IDNF if: it is in BCNF with respect to the functional dependencies; all inclusion dependencies are really foreign keys; and the set of inclusion dependencies is acyclic[1]. This normal form implies no interaction between the FDs and the INDs, prevents the problem of attribute redundancy, and satisfies a generalised form of entity integrity.

These normal forms are the goal in database design. As a result, most FDs and INDs occuring in practice are based on keys and foreign keys.

Definition 2.1.2 (Functional dependency). A *functional dependency* (FD) over a relation schema R is a statement of the form $R : X \to Y$, where $X, Y \subseteq R$ are sets of attributes.

An FD $R : X \to Y$ is satisfied in a relation r over R if whenever two tuples have

[1]The authors employ different terms: they state that a database is in IDNF if it is in BCNF w.r.t. the FDs, and if the set of inclusion dependencies is non-circular and key-based.

equal X-values, they also have equal Y-values. Formally, a functional dependency $R : X \to Y$ is *satisfied* (or holds) in a relation r over R, denoted by $r \models R : X \to Y$, if $\forall t_1, t_2 \in r,\ t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$.

An FD $R : X \to Y$ is *minimal* in r if Y is not functionally dependent on any proper subset of X, i.e., if $Z \to Y$ does not hold in r for any $Z \subset X$.

In Example 2.1.1 (page 9), Schedule: Theater Movie $\longrightarrow$ Time. This means that, in the theaters described in the MovieDB database, a movie is shown once a day, always at the same time.

Definition 2.1.3 (Inclusion dependency). An *inclusion dependency* (IND) over a database schema $\mathbf{R}$ is a statement of the form $R_1[X] \subseteq R_2[Y]$, where $R_1, R_2 \in \mathbf{R}$ and X, Y are sequences of attributes such that $X \subseteq R_1$, $Y \subseteq R_2$, and $|X| = |Y|$. A *unary inclusion dependency* (UIND) is an IND such that $|X| = |Y| = 1$. An IND $R_1[X] \subseteq R_2[Y]$ is of size i if $|X| = |Y| = i$.

Let d be a database over a database schema $\mathbf{R}$, where $r_1, r_2 \in d$ are relations over relation schemas $R_1, R_2 \in \mathbf{R}$. An inclusion dependency $R_1[X] \subseteq R_2[Y]$ is *satisfied* (or holds) in a database d over $\mathbf{R}$, denoted by $d \models R_1[X] \subseteq R_2[Y]$, if $\forall t_1 \in r_1, \exists t_2 \in r_2$ such that $t_1[X] = t_2[Y]$. (Equivalently, $d \models R_1[X] \subseteq R_2[Y]$ whenever $\pi_{r_1}(X) \subseteq \pi_{r_2}(Y)$.)

Note that X and Y are *sequences* of attributes: the ordering of the attributes within the sets X and Y matters. Indeed, the two following INDs are not equivalent: Manager[Firstname, Lastname] $\subseteq$ Employee[Firstname, Lastname] is different from Manager[Lastname, Firstname] $\subseteq$ Employee[Firstname, Lastname].

In Example 2.1.1, Schedule[Theater] $\subseteq$ Theaters[Name] and Schedule[Movie] $\subseteq$ Movies[Title]. The first inclusion dependency describes the fact that all theaters that show movies

(i.e., the theaters in Table Schedule) appear in the list of the town's theaters (i.e., are listed in Table Theaters). The second IND describes the fact that information such as length and genre (i.e., information displayed in Table Movies) is available for all movies currently playing (i.e., the movies in Table Schedule).

2.1.3 Dependency inference problem

Much work has been done on the implication problem of functional and inclusion dependencies [CFP84, LL01], but this is not our focus in this thesis. The implication problem for functional and inclusion dependencies is the problem of deciding, for a given set S of FDs and INDs, whether S logically implies α, where α is an FD or an IND.

Rather, we are interested in the *inference* problem of data dependencies: given a database d, find a small cover of all data dependencies satisfied in d, where the data dependencies can be functional dependencies [KM92], or inclusion dependencies [MLP02]. A cover for a set F of dependencies is a set G of dependencies such that F and G are equivalent, i.e., any database that satisfies all the dependencies of F also satisfies all the dependencies of G and vice versa [KM92]. A cover F is minimum if there is no cover G of F such that G is a subset of F and G has fewer dependencies than F. In the inference problem, the goal is to find a small but not necessarily minimum cover. Deducing a minimum cover out of a small cover could be added as a final step, independent of the data.

Most of the work on this research area has been devoted to only functional dependencies, both exact [HKPT98, NC01, LPL00, BMT89, BB95, Bel95, FS99] and approximate [HKPT98, KM92, Bou00], while inclusion dependencies have raised little

interest [MLP02, KMRS92, PT99, LPT99, BB95, Bou00].

Extracting functional or inclusion dependencies from a database instance has been shown to be hard in the worst case [MR87, KMRS92, MR92]:

- Mannila and Räihä proved that there are small relations where the number of functional dependencies is exponential with respect to the number of attributes of the relations [MR87]. The theorem is expressed as "For each n, there exists a relation r over R such that $n = |R|$, $|r| = O(n)$ and each cover of all functional dependencies has $\Omega(2^{n/2})$ dependencies." Therefore, the inference problem for FDs is inherently exponential with respect to the number of attributes.

- Assuming two n-attribute relations r_1, r_2, there are more than $n!$ possible nonequivalent inclusion dependencies from r_1 to r_2 [KMRS92]. The factorial comes from the fact that *sequences* and not sets of attributes are considered in INDs.

- Kantola et al. proved that what they call FULLINDEXISTENCE is an NP-complete problem in the size of the schema [KMRS92]. They define FULLINDEXISTENCE as follows: "Let R and S be relation schemas, and X a sequence consisting of the attributes of R in some order. Given relations r and s over R and S respectively, decide whether there exists a sequence Y consisting of disjoint attributes of S in some order such that the dependency $R[X] \subseteq S[Y]$ holds in the database (r, s)." Therefore, it is not always possible to check the existence of long INDs quickly.

Although our work is not about the implication problem, we can use inference rules for dependency implication in the inference problem, in order to speed up dependency discovery. We now give inference rules for FDs and INDs.

The following inference rules for the FD implication problem, known as Armstrong's axiom system [Arm74], form a sound and complete axiomatization of FDs:

1. (reflexivity): if $Y \subseteq X \subseteq R$, then $X \rightarrow Y$.

2. (augmentation): if $X \rightarrow Y$ and $W \subseteq R$, then $XW \rightarrow YW$.

3. (transitivity): if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

If X is a key for R, then the functional dependency $X \rightarrow R$ holds. So, by rule 2, the FDs $XW \rightarrow R$ also hold for all $W \subseteq R$, and thus there is no need to check these FDs against the database.

Casanova et al. give a sound and complete axiomatization of INDs, composed of the following inference rules [CFP84]:

1. (reflexivity): $R[X] \subseteq R[X]$, if X is a sequence of distinct attributes of the relation schema R.

2. (projection and permutation): if $R[A_1...A_m] \subseteq S[B_1...B_m]$, then $R[A_{i_1}...A_{i_k}] \subseteq S[B_{i_1}...B_{i_k}]$ for each sequence $i_1, ..., i_k$ of distinct integers from $\{1, ..., m\}$.

3. (transitivity): if $R[X] \subseteq S[Y]$ and $S[Y] \subseteq T[Z]$, then $R[X] \subseteq T[Z]$.

Thus, if the IND Manager[Firstname, Lastname] $\subseteq$ Employee[Firstname, Lastname] holds, then there is no need to test Manager[Lastname, Firstname] $\subseteq$ Employee[Lastname, Firstname] since we know from Rule 2 that it also holds. Again, if $S[Y] \subseteq T[Z]$ and $R[X] \not\subseteq T[Z]$, then by Rule 3, $R[X] \not\subseteq S[Y]$. We can also use the contrapositive of Rule 3 to avoid some tests: if $R[X] \subseteq S[Y]$ and $R[X] \not\subseteq T[Z]$, then $S[Y] \not\subseteq T[Z]$.

2.1.4 Approximate dependency inference problem

The *approximate dependency inference problem* is defined as the dependency inference problem in which the results do not need to be completely accurate: the dependencies found hold exactly or almost hold in the database [KM92]. We allow some false positives in the result, but no false negative. This means that some dependencies that do not hold exactly but almost hold are included in the result, and no dependency that holds exactly is missing from the result.

Kivinen and Mannila [KM92] concentrate on functional dependencies. To define formally a functional dependency that "almost holds", they present three measures to compute the error of an FD. The error measure retained by subsequent work on approximate functional dependency inference [HKPT98, Bou00] is the so-called g_3 measure, which is defined below.

The *error measure* g_3 of an FD $X \to Y$ in a relation r is the minimal fraction of tuples to remove from r for $X \to Y$ to hold [KM92]:

$$g_3(X \to Y, r) = 1 - \frac{\max\{\, |s| \mid s \subseteq r \text{ and } s \models X \to Y \,\}}{|r|}$$

Definition 2.1.4 (Approximate functional dependency). [KM92]
An *approximate FD* $X \to Y$ holds in a relation r with respect to an error threshold ϵ if and only if $g_3(X \to Y, r) \leq \epsilon$.
When the dependency holds exactly, the error g_3 is equal to 0, and when the dependency is violated in many tuples, the error tends to 1.

Computing the error for a given FD can be done in $O(|r|)$ time by using special data structures such as hash tables [HKPT98], or in $O(|r| \log r)$ time by sorting the data. But the computation is not always necessary since the error can be estimated

through upper and lower bounds.

Note that the error measure g_3 is a parameter of this definition of approximate FD, hence the definition remains the same even if we change the error measure.

The error measure g_3 defined for FDs has been adapted to the case of INDs in two different ways:

1. g_3 becomes the proportion of rows to remove from the left-hand-side relation for the IND to hold in a database d [Bou00, PT99]:

$$g_3(R_i[X] \subseteq R_j[Y], d) = 1 - \frac{\max\{\ |s| \mid s \subseteq r_i \text{ and } (d \setminus r_i) \cup s \models R_i[X] \subseteq R_j[Y]\}}{|r_i|}$$

2. Lopes et al. [LPT02] point out that this definition does not respect a natural property of INDs, namely that for an IND holding in a database, the number of distinct values of the left hand side is smaller than the number of distinct values of the right hand side. This is due to the fact that the definition of INDs is based on distinct tuples, while the error measure g_3 is not. Thus, they propose a new error measure for INDs, g'_3, which is the proportion of tuples with distinct X-values (independently of their occurrences) to remove from the left-hand-side relation for the IND to hold.

$$\begin{aligned} & g'_3(R_i[X] \subseteq R_j[Y], d) \\ & = 1 - \max\{\ |\pi_X(s)| \mid s \subseteq r_i \text{ and } (d \setminus r_i) \cup s \models R_i[X] \subseteq R_j[Y]\} \ / \ |\pi_X(r_i)| \\ & = 1 - |\pi_X(r_i) \bowtie_{X=Y} \pi_Y(r_j)| \ / \ |\pi_X(r_i)| \end{aligned}$$

This error measure is also used by Shen et al. in the form of a fitness criterion equal to $1 - g'_3$ [SZWA99].

We adopt in this thesis the second error measure g'_3, which better conveys the semantics of inclusion dependencies.

Definition 2.1.5 (Approximate inclusion dependency). An *approximate IND* $R_i[X] \subseteq R_j[Y]$ holds in database d, with respect to an error threshold ϵ, if and only if $g'_3(R_i[X] \subseteq R_j[Y], d) \leq \epsilon$.
Here again, we could replace the error measure g'_3 by any other measure.

2.1.5 Keys and foreign keys

In this thesis, we concentrate on keys and foreign keys, which are specializations of FDs and INDs respectively.

A superkey is a set of attributes that determines all the attributes in a relation schema, and a key is a superkey whose set of attributes is minimal.

Definition 2.1.6 (Key and approximate key). A set of attributes $K \subseteq R$ is a *superkey* for R if K functionally determines all other attributes of the relation, i.e., $R : K \rightarrow R$. A *key* (or minimal key) is a superkey such that no subset of it is a superkey.

K is a superkey in a relation r if no two tuples in r agree on K, thus K identifies uniquely each tuple in the relation.

Because a key is a special case of FD, the error measure for keys is the same as for FDs: the minimal fraction of tuples to remove for the dependency to hold in r. This simplifies into the fraction of tuples with duplicate K-values.

$$g_3(K, r) = 1 - \frac{|\pi_K(r)|}{|r|}$$

where $|\pi_K(r)|$ is the number of distinct K-values in r.

An *approximate key* K for r w.r.t. error threshold ϵ is a key such that $g_3(K, r) \leq \epsilon$.

At the schema level, a key is a constraint, but at the relation level, a key is a property derived from the relation. This corresponds to the distinction between a functional dependency (constraint) and the satisfaction of an FD in a relation (property).

For the specific database of Example 2.1.1 (page 9), that is, at the relation level, Title is a key for Movies, Name is a key for theaters, and {Theater, Movie} is a key for Schedule. Movie is an approximate key for Schedule w.r.t. error threshold 1/3; and Time is an approximate key for Schedule w.r.t. error threshold 1/6.

The number of keys in a relation can be exponential in the size of the schema:

Theorem 2.1.1. *[LL99] The number of keys for a schema R such that $|R| = n$ is at most*

$$\binom{n}{\lfloor n/2 \rfloor} = O\left(\frac{2^n}{\sqrt{n}}\right)$$

and there exists a relation r over R such that there exists $\binom{n}{\lfloor n/2 \rfloor}$ keys that are satisfied in r.

Definition 2.1.7 (Foreign key and approximate foreign key). Let $\mathbf{R}$ be a database schema, R_1, R_2 be relation schemas over $\mathbf{R}$, and let K be a key of R_2. In addition, let $d = \{r_1, r_2, ..., r_n\}$ be a database over $\mathbf{R}$.

A *foreign key* constraint is a specification of a set of attributes $X \subseteq R_1$ and a key K of R_2. The set of attributes X is called a *foreign key* (FK) of R_1 and it is said to *reference* the key K of R_2. The foreign key constraint that X references K is satisfied in d if the following condition holds: for all tuples $t_1 \in r_1$, there exists $t_2 \in r_2$

such that $t_1[X] = t_2[K]$. A foreign key is an IND whose right-hand side is a key. It is also called referential integrity constraint, or key-based inclusion dependency.

The error measure for foreign keys is thus the same as for INDs, g'_3. An *approximate foreign key* $R_1[X] \subseteq R_2[K]$ is satisfied (or holds) in d w.r.t. error threshold ϵ if $g'_3(R_1[X] \subseteq R_2[K], d) \leq \epsilon$.

In Example 2.1.1 (page 9), the following foreign keys hold: Schedule[Movie] $\subseteq$ Movies[Title], Theaters[Name] $\subseteq$ Schedule[Theater], and Schedule[Theater] $\subseteq$ Theaters[Name].

2.2 Review of literature

First of all, we present the levelwise algorithm used in a wide variety of data mining applications, from association rules to functional and inclusion dependencies. Then, we review the methods used in functional dependency or key discovery, and finally, the approaches for inclusion dependency or foreign key discovery.

2.2.1 Levelwise algorithm

One of the basic problems in knowledge discovery in databases (KDD) is to find all potentially interesting sentences from a dataset. Then, a user can select the really interesting ones. For example, these interesting sentences can be association rules, functional or inclusion dependencies. Mannila and Toivonen [MT97] study thoroughly a breadth-first or levelwise algorithm, also called generic data mining algorithm, for finding all potentially interesting sentences. Their paper includes a complexity analysis, as well as some applications, including functional and inclusion dependency discovery. Algorithms for discovering functional and inclusion dependencies within this

framework are also described elsewhere [Bou00]. The levelwise algorithm has been used, among other applications, for discovering association rules [AS94, MTV94], for discovering functional dependencies [HKPT98, LPL00, NC01], and for discovering inclusion dependencies [MLP02]. In this thesis, we use it to extract keys.

We define now more precisely the KDD framework of Mannila and Toivonen. Given a database d, define a language $\mathcal{L}$ for expressing properties of the data and a selection predicate q. The selection predicate q evaluates whether a sentence $\varphi \in \mathcal{L}$ defines a potentially interesting property of d. The task is to find a theory of d with respect to $\mathcal{L}$ and q, i.e., the set $\mathcal{T}h(\mathcal{L}, d, q) = \{\varphi \in \mathcal{L} \mid q(d, \varphi) \text{ is true}\}$. Consider the example of inclusion dependency discovery. The language $\mathcal{L}$ consists of all inclusion dependencies, and the predicate q is simply the satisfaction predicate.

To apply the levelwise algorithm, we need to define a specialization relation $\preceq$ between sentences (i.e., a partial order), which is also monotonic w.r.t q. That is, if $\varphi \preceq \theta$ (i.e., θ is more specific than φ) and $q(\theta, d)$ is true, then $q(\varphi, d)$ is also true. This means that if a sentence θ is potentially interesting w.r.t q, then all more general sentences φ are also potentially interesting w.r.t q.

In our example of IND discovery, the specialization relation $\preceq$ is defined as follows: if $\varphi = R[X] \subseteq S[Y]$ and $\theta = R'[X'] \subseteq S'[Y']$, we have $\varphi \preceq \theta$ if $R = R', S = S'$ and the sequences X and Y are contained in X' and Y' respectively. For instance, $R[A] \subseteq S[D] \preceq R[ABC] \subseteq S[DEF]$.

Then, the idea in the levelwise algorithm is to start from the most general sentences and try to generate and evaluate more and more specific sentences, but without evaluating sentences that we know from previous stages will not be interesting. This corresponds to pruning the search space. In the case of IND discovery, we are interested in INDs with the longest attribute sequences, so we start from INDs with small

sequences, and step by step, we add attributes until the longest ones are found. But when for instance $R[A] \subseteq S[D]$ is not satisfied, we know that $R[ABC] \subseteq S[DEF]$ cannot be satisfied, so we do not evaluate it. For some languages, such as functional and inclusion dependencies, it is possible to compute the interesting sentence candidates at one level uniquely from the candidates and interesting sentences at the previous level, thus saving memory.

This framework, when applied as such to key discovery, searches for superkeys starting from the large ones and moving towards small ones, finding eventually the keys. But as keys tend to be of small size, this algorithm is highly inefficient. Another algorithm, also based on the principles of exploring a search space level by level, reusing the results of previous levels, and pruning the search space as soon as possible, has been proposed by Huhtala et al. [HKPT98]. But unlike the framework described above, this algorithm does not rely on a specialization relation to direct the search. It is more suitable for FD/key discovery since it searches for FDs from small to large. Therefore we applied this algorithm to key discovery in this thesis. We describe informally in Figure 2.1 the levelwise algorithm alongside the algorithm applied to key discovery.

Note that this algorithm also works for approximate inference. It suffices that the selection predicate q evaluates to true when a sentence φ almost holds.

To summarize, the main advantages of this algorithm are twofold: reducing the computation at each level by reusing the results of previous levels, and efficiently pruning the search space.

Levelwise algorithm for finding all potentially interesting sentences.

Input: a database d (consisting of only one relation r over R for the key discovery example), a language $\mathcal{L}$ with specialization relation $\preceq$, and a selection predicate q.

Output: $\mathcal{T}h(\mathcal{L}, d, q)$.

Method:

	Key discovery example
level := 1	
$C(1) :=$ most general candidates in $\mathcal{L}$ w.r.t $\preceq$.	$C(1) := \{A \mid A \in R\}$
while $C(level) \neq \emptyset$ do	
$F(level) := \{ \varphi \in C(level) \mid q(d, \varphi)\}$	$Keys(level) := \{X \in C(level) \mid$ $\lvert\pi_X(r)\rvert = \lvert r\rvert\}$
$C(level+1) :=$ genNextCand$(C(level), F(level))$	$C(level+1) := \{X \mid$ $\lvert X\rvert = level+1,$ X is not a superset of a key in $Keys(level)\}$
// $\forall \varphi \in C(level), \theta \in C(level+1),\ \varphi \prec \theta$	
level := level + 1	
done	
output $\cup_{each\ level} F(level)$	output $\cup_{each\ level} Keys(level)$

Figure 2.1: Levelwise algorithm

2.2.2 Key and functional dependency discovery

The most efficient methods to determine (approximate) minimal functional dependencies, TANE [HKPT98], Dep-Miner [LPL00], FUN [NC01], each use an instance of the levelwise algorithm. All of these approaches work with a much smaller representation of a relation called a stripped partition database, from which the FDs can be derived efficiently. A partition of a relation r under an attribute set X is a set of equivalence classes, where an equivalence class is the set of tuples in r sharing the same value for the attributes in X. A stripped partition is a partition in which the equivalence classes of size one have been removed. The idea underlying this removal is that, when an equivalence class consists of a unique tuple, this tuple does not share the value of the considered attributes with any other tuple, and therefore, this tuple cannot break any functional dependency. The new representation of r: the stripped partition database $\widehat{r}$, is the union of the stripped partitions for all attributes in the relation schema. Processing the data by using stripped partitions allows them to perform linearly w.r.t. the number of tuples in the relation. However, constructing the partitions in the first place requires special data structures such as hash tables in order to be done linearly w.r.t. the number of tuples $|r|$, or it can be performed by sorting the database, which is in $O(|r| \log |r|)$.

The three methods cited above differ in their characterization of FDs, i.e., in how they generate FD candidates and evaluate them. We now describe in more detail each of these methods, using the database of Example 2.2.1.

Example 2.2.1. Consider the database composed of the following relation r only (taken from [LPL00]). The attribute names have been assigned a single letter for briefness, and will be designated by this letter in the rest of this section.

Tuple No.	empnum A	depnum B	year C	depname D	mgr E
1	1	1	1985	Biochemistry	5
2	1	5	1994	Admission	12
3	2	2	1992	Computer Science	2
4	3	2	1998	Computer Science	2
5	4	3	1998	Geophysics	2
6	5	1	1975	Biochemistry	5
7	6	5	1988	Admission	12

From now on, we represent a tuple by its tuple number (the value in the column Tuple No.).

The partition of r under A is $\pi_A = \{\{1,2\},\{3\},\{4\},\{5\},\{6\},\{7\}\}$. The stripped partition of r under A is $\widehat{\pi_A} = \{\{1,2\}\}$. Similarly, we obtain $\widehat{\pi_B} = \{\{1,6\},\{2,7\},\{3,4\}\}$, $\widehat{\pi_C} = \{\{4,5\}\}$, $\widehat{\pi_D} = \{\{1,6\},\{2,7\},\{3,4\}\}$, and $\widehat{\pi_E} = \{\{1,6\},\{2,7\},\{3,4,5\}\}$. The stripped partition database is $\widehat{r} = \{\{1,2\},\{1,6\},\{2,7\},\{3,4\},\{3,4,5\}\}$.

Tane

The goal in Tane [HKPT98] is to find all minimal (approximate) FDs holding in a relation. The algorithm searches for FDs of the form $X \setminus \{A\} \to A$ by examining attribute sets X of increasing sizes: first, sets of size 1, then of size 2, and so on, using an instance of the levelwise algorithm. For each attribute set X considered, it constructs a collection of potential right-hand sides, $C(X)$. An attribute A is in $C(X)$ provided that A is not determined by any proper subset of X which does not contain A; this enforces the minimality of the FD[2]. The set $C(X)$ can be computed from the

[2]The algorithm actually uses a more powerful version of these sets $C(X)$, with more pruning, but to give the idea of the algorithm, we limit ourselves to the simple version.

candidate sets of X's subsets, as the intersection of the sets $C(X \setminus \{A\})$ for all $A \in X$. Then, the algorithm tests the potential FDs $X \setminus \{A\} \rightarrow A$ for all $A \in X \cap C(X)$. When the FD is satisfied, it prunes A from the set $C(X)$ to preserve the minimality of future FDs. When all attribute sets X of size s have been processed, the algorithm generates the attribute sets of size $s + 1$ to be considered.

In Tane, testing FDs is based on partition refinement: the dependency $X \setminus \{A\} \rightarrow A$ holds if the partition $\pi_{X \setminus \{A\}}$ refines $\pi_{\{A\}}$, i.e., if every equivalence class in $\pi_{X \setminus \{A\}}$ is a subset of some equivalence class in $\pi_{\{A\}}$. Practically, the authors use the equation $e(X \setminus \{A\}) = e(X)$ to test whether the FD is satisfied, where $e(X)$ is the difference between the sum of the sizes of the equivalence classes in the stripped partition $\widehat{\pi_X}$ and the number of such equivalence classes. This equation allows them to deal with stripped partitions, and to limit the amount of memory used by the algorithm since they exploit results from the previous level only.

On the database of Example 2.2.1, the algorithm would unfold as shown in Figure 2.2.

Dep-Miner

In Dep-Miner [LPL00], the underlying idea is based on the concept of *agree set*, which groups all attributes having the same value for a given pair of tuples. From these sets, the authors derive maximal sets. The maximal sets for some attribute A are the largest possible sets of attributes not determining A. Then, from the complements of these maximal sets, they derive the left-hand sides of FDs using a levelwise algorithm: for each attribute A, it searches for left-hand sides X such that $r \models X \rightarrow A$ by increasing the size of X. The only step that requires accessing the database (or rather, the stripped partition database) is the computation of agree sets.

The authors avoid computing agree sets for all pairs of tuples by limiting them-

X	$C(X)$	$\widehat{\pi_X}$	$e(X)$	$X \setminus \{A\} \stackrel{?}{\to} A$
$\emptyset$	$\{A, B, C, D, E\}$	$\emptyset$	0	
A	$\{A, B, C, D, E\}$	$\{\{1, 2\}\}$	1	$\emptyset \not\to A$
B	$\{A, B, C, D, E\}$	$\{\{1, 6\}, \{2, 7\}, \{3, 4\}\}$	3	$\emptyset \not\to B$
C	$\{A, B, C, D, E\}$	$\{\{4, 5\}\}$	1	$\emptyset \not\to C$
D	$\{A, B, C, D, E\}$	$\{\{1, 6\}, \{2, 7\}, \{3, 4\}\}$	3	$\emptyset \not\to D$
E	$\{A, B, C, D, E\}$	$\{\{1, 6\}, \{2, 7\}, \{3, 4, 5\}\}$	3	$\emptyset \not\to E$
AB	$\{A, B, C, D, E\}$	$\emptyset$	0	$A \not\to B$, $B \not\to A$
AC	$\{A, B, C, D, E\}$	$\emptyset$	0	$A \not\to C$, $C \not\to A$
$\vdots$				
BD	$\{A, B, C, D, E\}$	$\{\{1, 6\}, \{2, 7\}, \{3, 4\}\}$	3	$B \to D$, $D \to B$
$\vdots$				
ABC	$\{A, B, C, D, E\}$	$\emptyset$	0	$AB \to C$, $AC \to B$, $BC \to A$
$\vdots$				

Figure 2.2: Unfolding of Tane on the database of Example 2.2.1

selves to the tuples within MC, the set of maximal equivalence classes of the stripped partition database: $MC = max_{\subseteq}\{c \in \widehat{\pi} \mid \widehat{\pi} \in \widehat{r}\}$; $MC = \{\{1,2\},\{1,6\},\{2,7\},\{3,4,5\}\}$ in our example. An attribute A is included in the agree set of tuples (t_1, t_2) if t_1 and t_2 belong to the same equivalence class in the stripped partition $\widehat{\pi_A}$. In our example, the agree set for the pair of tuples (1,2) is $ag(1,2) = \{A\}$. Similarly, we have $ag(1,6) = ag(2,7) = ag(3,4) = \{B,D,E\}$, $ag(3,5) = \{E\}$, $ag(4,5) = \{C,E\}$, so the agree sets of r are $ag(r) = \{A, BDE, E, CE\}$.

The authors derive the maximal sets from the agree sets as follows: for an attribute A, the maximal set $max(A,r) = max_{\subseteq}\{X \in ag(r) \mid A \notin X, X \neq \emptyset\}$. In our example, we have $max(A,r) = \{BDE, CE\}$ and the complement of the maximal set of A is $cmax(A,r) = \{AC, ABD\}$.

Finally, they derive the left-hand sides of FDs from these complements of maximal sets. The algorithm is based on the characterization of left-hand sides of FDs of the form $LHS \rightarrow A$ as the set of minimal transversals of the simple hypergraph $cmax(A,r)$[3]. For instance, we have $cmax(A,r) = \{AC, ABD\}$; the set $\{B,C\}$ is a transversal of $cmax(A,r)$ since $\{B,C\} \cap \{A,C\} \neq \emptyset$, and $\{B,C\} \cap \{A,B,D\} \neq \emptyset$. In a levelwise manner, the algorithm proceeds by increasing sizes of candidate transversals. If a candidate T of size s is a transversal, then no superset of T is allowed among the candidates of size $s+1$. The candidates of size 1 are the attributes in $cmax(A,r)$.

The authors found their approach to outperform Tane in all data configurations tested in the experiments.

On the example, the algorithm unfolds as shown in Figure 2.3.

[3]A collection $\mathcal{H}$ of subsets of R is a simple hypergraph if
$\forall X \in \mathcal{H},\ X \neq \emptyset$ and $(X, Y \in \mathcal{H}$ and $X \subseteq Y \ \Rightarrow X = Y)$.
A transversal T of $\mathcal{H}$ is a subset of R such that $\forall E \in \mathcal{H}, T \cap E \neq \emptyset$.

		Size 1		Size 2	
RHS	$cmax(RHS, r)$	Candidates	Transversals (LHS)	Candidates	Transversals (LHS)
A	$\{AC, ABD\}$	A, B, C, D	A	BC, BD, CD	BC, CD
B	$\{BCDE, ABD, ABCD\}$	A, B, C, D, E	B, D	AC, AE, CE	AC, AE
C	$\{BCDE, AC, ABCD\}$	A, B, C, D, E	C	AB, AD, AE, BD, BE, DE	AB, AD, AE
D	$\{BCDE, ABD, ABCD\}$	A, B, C, D, E	B, D	AC, AE, CE	AC, AE
E	$\{BCDE\}$	B, C, D, E	B, C, D, E	$\emptyset$	-

Figure 2.3: Unfolding of Dep-Miner on the database of Example 2.2.1

So the non-trivial FDs satisfied in r are the following:
$BC \rightarrow A$, $CD \rightarrow A$, $D \rightarrow B$, $AC \rightarrow B$, $AE \rightarrow B$, $AB \rightarrow C$, $AD \rightarrow C$, $AE \rightarrow C$, $B \rightarrow D$, $AC \rightarrow D$, $AE \rightarrow D$, $B \rightarrow E$, $C \rightarrow E$, $D \rightarrow E$.

FUN

In FUN [NC01], the authors describe their approach at a general level only, without detailing the optimizations due to the stripped partition database representation, even though they use this representation in their implementation.

Their characterization of FDs is based on the concept of *free sets*. A free set X is a set of attributes such that removing an attribute from X decreases the number of X's distinct values[4]. Free sets correspond to left-hand sides of FDs. To characterize

[4]Formally, X is a free set if $\nexists Y \subset X$ such that $|\pi_Y(r)| = |\pi_X(r)|$.

right-hand sides of FDs, they define the closure and quasi-closure of an attribute set X. The closure of X, denoted by X^+, is the union of the attributes in X and the additional attributes which can be added to X without increasing the number of X's distinct values: $X^+ = X \cup \{A \in R \setminus X \mid |\pi_X(r)| = |\pi_{X \cup A}(r)|\}$. The quasi-closure of X, $X^\diamond$, is the union of X and the closures of X's maximal subsets. While all attributes in X^+ are determined by X, only those not in X's quasi-closure yield minimal FDs. In summary, the minimal FDs satisfied in r are the FDs of the form $X \rightarrow A$ where X is a free set and $A \in X^+ \setminus X^\diamond$. This approach relies heavily on counting the number of distinct values of attribute sets, which requires either a sort of the data, or special data structures; but while the latter solution is used by the authors, they do not detail how they perform the counting operation efficiently.

In a levelwise manner, the algorithm searches for free sets of increasing sizes. At the level corresponding to FDs with a left-hand side of size s, the algorithm knows from the previou s level the free sets of size s and their quasi-closure, as well as the collection of candidate free sets of size $s+1$. It first computes the closure of the free sets of size s, and displays the FDs of the form $X \rightarrow A$ where X is a free set of size s and $A \in X^+ \setminus X^\diamond$. Then, it computes the quasi-closure of the candidate free sets of size $s+1$, using the closure of the free sets of size s. Then, it prunes the candidate free sets X of size $s+1$ that are not free sets, based on the number of distinct values of X and of its maximal subsets that are free sets[5]. Finally, it generates the candidate free sets of size $s+1$ from the free sets of size s. The authors found that their approach outperforms Tane in all configurations investigated, which they explain by the fact that the number of FDs tested in their approach is less than in Tane.

[5]These two steps, computing quasi-closures and pruning candidate free sets, are presented in this order in the paper. However, it seems more efficient to perform them in the reverse order in order to avoid computing the quasi-closures of candidate free sets that are not free sets.

On the example, the algorithm would unfold as shown in Figure 2.4.

X	$\lvert\pi_X(r)\rvert$	$X^\diamond$	X^+	$X \to A$
A	6	A	A	-
B	5	B	B, D, E	$B \to D, E$
C	6	C	C, E	$C \to E$
D	4	D	D, B, E	$D \to B, E$
E	3	E	E	-
AB	7 (key)	A, B, D, E	A, B, C, D, E	$AB \to C$
AC	7 (key)	A, C, E	A, B, C, D, E	$AC \to D, E$
AD	7 (key)	A, B, D, E	A, B, C, D, E	$AD \to C$
AE	7 (key)	A, E	A, B, C, D, E	$AE \to B, C, D$
BC	7 (key)	B, C, D, E	A, B, C, D, E	$BC \to A$
BD	5 (not a free set)	-	-	-
BE	5 (not a free set)	-	-	-
CD	7(key)	B, C, D, E	A, B, C, D, E	$CD \to A$
CE	6 (not a free set)	-	-	-
DE	4 (not a free set)	-	-	-

Figure 2.4: Unfolding of FUN on the database of Example 2.2.1

Key discovery

The approaches focusing only on key discovery [KA96, SZWA99] also use the levelwise algorithm. Shen et al. [SZWA99] implemented an algorithm with the data being stored in a DBMS, thus querying the database to test the validity of key candidates.

However, they do not provide the details of the algorithm they implemented. Knobbe and Adriaans [KA96] search for keys of increasing sizes. An attribute set X is a key if the number of distinct values of X is equal to the number of tuples in the relation. When X is not a key, they record the set $Black(r, X)$ of tuples having duplicate X-values. Then, when they test whether XA is key, they only need to sort the tuples in $Black(r, X)$ w.r.t. XA and check that they all have different XA-values. In such a case, XA is key, otherwise, they record $Black(r, XA)$ and proceed to the next level of the algorithm. As the sets $Black(r, X)$ can have a size in $O(|r|)$, the amount of memory required for these structures makes their method impractical for large databases.

Our approach for key discovery is adapted from Tane. We dropped the right-hand side candidate sets, and perform key testing via database queries instead of using the stripped partition database. But the rest is similar to Tane: we search for keys in a levelwise manner, by increasing size. At each level of the algorithm, we test the key candidates, then prune the search space, and then generate the candidates for the next level.

In this thesis, we do not need elaborate data structures since we keep the data inside the DBMS. Thus, even though our algorithms may not be as efficient w.r.t. the number of tuples as those of FD discovery presented above, the amount of memory used by the algorithm is kept very low.

2.2.3 Foreign key and inclusion dependency discovery

Unlike functional dependencies, extracting inclusion dependencies (or foreign keys) has raised little interest in the database community. De Marchi et al. [MLP02]

explain this disregard by the complexity of the problem (see Section 5.3), and a lack of popularity.

Due to the huge number of possible INDs [PT99], it is impractical to enumerate all inclusion dependency candidates and try to evaluate them on a database. Existing work focuses on how to prune the number of inclusion dependency candidates considered.

One approach [LPT02, PT99] consists of selecting a much reduced initial set of inclusion dependency candidates by considering only the attributes appearing together in equi-joins from a workload of query statements. Thus, this approach requires such a workload of queries to the database. The authors also claim that this criterion allows them to find out only "interesting" dependencies, since they were used in practice. But their method is not complete: they determine only a subset of all dependencies, which depends on some past queries to the database.

Another approach is to start with unary inclusion dependencies. Indeed, there are only a polynomial number of them (n^2 in a database with n attributes). So the complexity of extracting the unary INDs for a database with n attributes and p tuples is polynomial with respect to both parameters: it is $O(n^2 p \log p)$ [MR92, KMRS92]. Moreover, unary INDs are a necessary condition for non-unary (or composite) INDs: for $R[AB] \subseteq S[CD]$ to hold, we must have $R[A] \subseteq S[C]$ and $R[B] \subseteq S[D]$. We chose this approach because it does not rely on any prior knowledge on the database.

The problem of efficiently extracting unary inclusion dependencies has been studied in two different contexts:

1. all data processing is done via database queries [BB95], or

2. a new representation of the data is generated, and the extracting step deals only

with this new representation [MLP02].

In the first context, Bell and Brockhausen [BB95] optimize their algorithms to minimize the number of accesses to the database. They distinguish two generic types: number and string, and determine UINDs for attributes of each type independently. To restrict the number of UIND candidates, they use value restrictions, i.e., upper and lower bounds on the attribute domains, as follows: the bounds of a left-hand side must lie in the interval made by the bounds of the right hand side. When testing the UIND candidates against the database, they exploit the transitivity of UINDs to reduce the number of queries, using the two following rules:

1. $A_i \subseteq A_j \; and \; A_k \not\subseteq A_j \Longrightarrow A_k \not\subseteq A_i$
2. $A_i \subseteq A_j \; and \; A_i \not\subseteq A_k \Longrightarrow A_j \not\subseteq A_k$

Depending on the ordering of the attributes, the transitivity properties can save more or fewer accesses to the database, but as long as there is at least one valid UIND in the database, their algorithm saves at least one database query.

In the second context, De Marchi et al. [MLP02] also determine UINDs for each data type independently. For a given data type t, they build a binary relation associating, for each value of type t appearing in the database, the attributes in which it appears. Thus the amount of memory required is proportional to the number of distinct values in the database. Note that all these binary relations can be generated in one pass over the database. Then, they deduce the UINDs from a binary relation $\mathbb{B}$ as follows: if A is included in C, then each value taken by A must be associated in the binary relation with C. Therefore, the right-hand sides of A are given by the intersection, for each value v of A, of the attribute sets associated in $\mathbb{B}$ with v. Here again, computing the intersections for all attributes with type t can be achieved in one pass over the binary relation for the type t.

De Marchi et al. compared both approaches on six synthetic databases, having small or medium size so the binary relations fit in main memory. Their approach was found to outperform Bell and Brockhausen's. However, the binary relations may not always fit in memory.

We now consider how the knowledge of UINDs can be used to find foreign keys or general inclusion dependencies.

Knobbe and Adriaans [KA96] propose to use UINDs to identify foreign keys. They first extract the keys, then the UINDs, then form the FK candidates, and then, test these candidates against the database. They use key-based UINDs to form the foreign key candidates $F = F_1...F_k$ for a given key $K = K_1...K_k$ by taking each F_i in the left-hand sides of the UINDs having K_i as a right-hand side. When there are several FK candidates, they suggest the use of heuristics to choose among them.

After having extracted UINDs by using binary relations (as explained above) instead of database queries, De Marchi et al. [MLP02] use an instance of the levelwise algorithm to determine the INDs in which the testing of INDs is performed via database queries. This instance of the levelwise algorithm devoted to INDs was described by Mannila and Toivonen [MT97] and Boulicaut [Bou00], but De Marchi et al. detail in their paper [MLP02] how to generate the IND candidates of size $i + 1$ from the INDs of size i.

Chapter 3

Algorithms and complexity

We present in this chapter a framework for discovering (approximate) keys and foreign keys holding in a database instance, of which we assume no previous knowledge. The problem tackled can be divided in two main modules: (1) identifying the (approximate) keys, and (2) using the keys found to extract the (approximate) foreign keys.

We also analyze the complexities of the algorithms proposed.

For simplicity reasons, we implemented the algorithms on top of a DBMS, but our framework could easily be adapted to use direct processing of the data.

3.1 General architecture

Our general architecture for discovering the approximate keys and foreign keys from a database, of which we assume no prior knowledge, is outlined in Algorithm 1.

The inputs of the algorithm are a database d and the approximation thresholds for keys and foreign keys. The algorithm outputs the keys and foreign keys which

Algorithm 1: General extraction of keys and foreign keys

Input : a database d; the maximum approximation degrees for keys and foreign keys, ϵ_{key} and ϵ_{fk} respectively; and the pruning parameter $PgParam$ (i.e., the fraction of tuples involved in the pruning pass)

Output: the "most interesting" keys and their associated foreign keys holding in d

% Preliminary step:

Extract statistics about the database

foreach *table* tab *in the database* **do**

    size ← 1

[1.1] **while** *StoppingCriterion(tab,size)* = *false* **do**

        % Find all the keys of size size.

        Key(size) ← ExtractKeys(tab,size,ϵ_{key})

        foreach *key* ∈ *Key(size)* **do**

            % Find the foreign keys and their degree of approximation.

            key.ForeignKeys ← ExtractForeignKeys(key,$\epsilon_{fk}, PgParam$)

        end

[1.2]   Sort Key(size) by interestingness and merge it with the sorted key list of tab.

        size ← size + 1

    end

    Output the key list of tab, including for each key its foreign keys.

end

almost hold in d w.r.t. the approximation thresholds.

As a preliminary step, we extract the following statistics about the database: the number of tuples of each table, and the type and number of distinct values of each attribute. These statistics will be used to optimize the different steps of the general algorithm.

Then, we process each table in the database in turn. For a given table, we determine the integrity constraints by increasing size: first, the keys and foreign keys (FKs) of size 1, then the keys and FKs of size 2, working towards larger sizes. To identify the integrity constraints of a given size s, we first extract the keys of size s (as detailed in Algorithm 2, see page 41), and then, process them in turn to find their FKs (as detailed in Algorithm 4, see page 46).

A complete algorithm, finding all the keys and foreign keys holding in the database, would process each table until all its keys and respective FKs are found. In such a case, the processing of a table would only stop when the maximal key size -the number of attributes of the relation- has been reached, or when there are no more candidates. So the stopping criterion [1.1] of the general algorithm (Algorithm 1) evaluates to true if and only if size is equal to the number of attributes of tab, or there is no key candidate.

But in most cases, we are only interested in a small subset of all integrity constraints (ICs). Typically, the most useful keys are referenced by many FKs, have small size and approximation degree, and their FKs also have a small approximation degree. Depending on the application for which keys and foreign keys are discovered, we may want to take into account the number of different relations containing foreign keys referencing a given key rather than just the number of foreign keys. Indeed, we

may prefer a key with two foreign keys in different relations to a key with foreign keys in the same relation. This criterion would favor integrity constraints that connect the different relations of a database, which would be useful in schema mapping [MHH00].

These different criteria defining the interestingness of ICs are not independent. For example, there is a tradeoff between the number of FKs and the size, so we may prefer a key of size 2 with 3 FKs to a key of size 6 with 4 FKs. Similarly, we may prefer a key of size 3 holding exactly in the dataset to an approximate key of size 2. Thus, in order to achieve the best ordering of the ICs, we need to extract them all, and only then can we order them by degree of interestingness. But finding them all when we only want a small subset of them is highly inefficient. In addition, the best ordering is very subjective and is hard to define accurately independently of the set of elements we want to order. So we decided to implement a heuristic algorithm that would find a good enough subset with a reasonable amount of work. We chose to mimic the complete algorithm on a smaller scale: we extract all keys and FKs of size bounded by a constant. This constraint reflects the fact that in practice, for efficiency reasons, referenced keys tend to be small. Indeed, otherwise, large parts of the data are duplicated. Then, because there may be many ICs within the bound chosen, we want to highlight the best of these. So we propose an order relation on integrity constraints, which reflects the general usefulness of such constraints as keys and foreign keys. This ordering was used by Shen et al. [SZWA99].

Definition 3.1.1 (Integrity constraint ordering by interestingness). The elements for the ordering are pairs formed of an approximate key K and the set $FKset$ (possibly empty) of approximate foreign keys referencing it. We denote by approx(K) (resp. approx(F)) the approximation degree of a key K (resp. foreign key F). To compare two sets of FKs, we define the average approximation degree of a FK set

$FKset$, denoted by average_approx($FKset$), as the average of the approximation degrees of all FKs included in $FKset$: average_approx($FKset$) = avg{ approx(F) | $F \in FKset$}.

Consider 2 elements $e1 = (K1, FKset1)$ and $e2 = (K2, FKset2)$. We define the ordering between them according to: (1) the number of FKs attached to the keys, (2) the sizes of the keys, (3) the approximation degrees of the keys, and (4) the average approximation degrees of the foreign keys, as follows:

$$
\begin{array}{lll}
e1 \prec e2 & \text{if} & |FKset1| < |FKset2| \\
 & \text{else if} & |FKset1| = |FKset2| \text{ and } size(K1) > size(K2) \\
 & \text{else if} & |FKset1| = |FKset2| \text{ and } size(K1) = size(K2) \text{ and} \\
 & & \text{approx}(K1) > \text{approx}(K2) \\
 & \text{else if} & |FKset1| = |FKset2| \text{ and } size(K1) = size(K2) \text{ and} \\
 & & \text{approx}(K1) = \text{approx}(K2) \text{ and} \\
 & & \text{average\_approx}(FKset1) > \text{average\_approx}(FKset2)
\end{array}
$$

For the tests, the upper bound on the size was fixed to 5 by Shen et al. [SZWA99]; we chose the upper bound on the size to be equal to 4 arbitrarily, because our test databases have a smaller schema than theirs.

To adapt the general architecture so that we only determine the most interesting (in the sense of Definition 3.1) integrity constraints of bounded size, we only have to change the stopping criterion [1.1] and add a possible pruning step after step [1.2]. This pruning step would be used to limit the number of integrity constraints discovered at each stage of the algorithm by removing the least interesting of them. For example, if we want to determine primary keys, as did Shen et al. [SZWA99], we can prune all ICs but one, the best, at each step (when there are several "best", we

can choose one randomly). Finally, the stopping criterion evaluates to true when the number and quality of the integrity constraints discovered up to that point match the specifications of the user, or when the size of the keys/FKs mined has reached an upper bound (in our experiments, the upper bound is fixed to 4).

3.2 Finding the keys

We want to extract the approximate keys of a given size holding in the database. The inputs of the algorithm are a relation r, the key size, and the approximation threshold ϵ_{key}. The algorithm outputs all approximate keys having the specified size that hold in r w.r.t. approximation threshold ϵ_{key}.

In the general algorithm (Algorithm 1, see 36), we search for keys by increasing sizes, starting with keys of size 1 and going towards larger sizes. Identifying the keys of a given size is done via a call to the function `ExtractKeys` (Algorithm 2). As soon as a key is found, we prune the search space so that we do not extract superkeys: if X is a key, then we discard from the search space all supersets of X.

Since keys are just a special case of functional dependencies, we adapted Tane's architecture [HKPT98] to our problem of finding the minimal keys, and to the fact that, unlike in Tane, we keep the data inside the database management system, which limits how we can process it. The resulting method is shown in Algorithm 2.

For each size, the key candidates are stored in $CandKey(size)$. We test them against the database, and insert the candidates that were found to be keys in $Key(size)$. The rest of the candidates are inserted in $CandNotKey(size)$. We then compute from *CandNotKey(size)* the set *CandKey(size+1)* of key candidates for the next level of

Algorithm 2: ExtractKeys

Input : a relation r with n attributes A_1, ..., A_n

$size \leq n$, the number of attributes of the keys to extract

ϵ_{key}, the approximation thresholds for keys

Output: the set $Keys(size)$ containing all the keys with $size$ attributes of r

if *size = 1* **then**

% Initialization

$CandKey(1) = \{A_1, ..., A_n\}$

$Key(1) = \{A \in CandKey(1) \mid |A| = |r|\}$

$Keys(r) \leftarrow Key(1)$

$CandNotKey(1) \leftarrow CandKey(1) - Key(1)$

$CandKey(2) \leftarrow$ `GenerateNextCand` $(CandNotKey(1))$

end

else

% $CandKey(size)$ has been computed by ExtractKeys$(r, size - 1, \epsilon_{key})$

foreach X *in* $CandKey(size)$, $X = X_1...X_{size}$ **do**

% card(X) has been initialized when generating the candidate X

[2.1] **if** $card(X) \geq (1 - \epsilon_{key}).|r|$ **then**

card$(X) \leftarrow$ `SELECT COUNT(*)`

`FROM (SELECT DISTINCT` $X_1, ..., X_p$ `FROM` r`)`

[2.2] **if** $card(X) \geq (1 - \epsilon_{key}).|r|$ **then**

% X is an (approximate) key

$Key(size) \leftarrow Key(size) \cup X$

end

end

end

$CandNotKey(size) \leftarrow CandKey(size) - Key(size)$

$CandKey(size + 1) \leftarrow$ `GenerateNextCand` $(CandNotKey(size))$

end

Output $Keys(size)$

the algorithm, at *size+1*.

The initial key candidates, of size 1, are the attributes of the relation. We treat size 1 as a special case because we do not need to query the database to determine the keys: we use the statistics gathered in the preliminary step.

3.2.1 Testing the key candidates

An attribute set X is an (approximate) key if it has no more than $\epsilon_{key}.|r|$ duplicate values, where ϵ_{key} is the approximation threshold and $|r|$ the number of rows of the relation, or equivalently, if it has at least $(1 - \epsilon_{key}).|r|$ distinct values ([2.2]). So to test this inequality, we query the database to retrieve X's number of distinct values, stored in card(X) (for cardinality). In some cases, we are able to determine that X cannot be a key without querying the database, when an upper bound on X's cardinality does not pass the test ([2.1]). This upper bound is computed when generating the candidate X (see next section), and is used as the initial value of card(X). When X does not pass the test, card(X) keeps its initial value, i.e., the upper bound on X's cardinality, but when X does pass the test, card(X) becomes X's real cardinality. The upper bound is based on the cardinalities of X's maximal subsets: $\max\{|A|.\text{card}(X \setminus \{A\}) \mid A \in X\}$.

In Tane [HKPT98], the use of bounds to avoid accessing the data is limited to potential FDs because these are the focus.

3.2.2 Generating key candidates

First of all, we show that we can compute the key candidates at iteration $size + 1$ only from $CandNotKey(size)$, the set of key candidates that were found not to be keys at iteration $size$.

Definition 3.2.1. An attribute set X is a *key candidate* if no proper subset of X is a key, otherwise X would not be a key but a superkey.
$CandKey(s) = \{X \text{ s.t. } |X| = s \text{ and } \forall Y \subset X,\ Y \text{ is not a key}\}$

This definition is the basis for the following proposition.

Proposition 1. *An attribute set X of size $s+1$ is a key candidate if all its subsets of size s belong to $CandNotKey(s)$.*

We adapted the function which generates the functional dependency candidates for the next level from Tane [HKPT98] to our particular case of keys.

In the function `GenerateNextCand` we generate the key candidates of size $s+1$ in two steps: (1) we construct attribute sets of size $s+1$, which is done at [3.1] and detailed below, and (2) we check that all their subsets of size s are included in $CandNotKey(s)$, which is done at [3.2].

In the first step, rather than generating all possible sets of size $s+1$, we start from attribute sets in $CandNotKey(s)$, and combine them to form sets of size $s+1$. Thus, part of the subset tests are already verified, therefore it is more likely that the generated attribute sets will be key candidates. To combine two sets Y, Z of size s in order to form a set X of size $s+1$, it is necessary that Y and Z share exactly $s-1$ attributes. Consider $X = X_1...X_{s+1}$, where each X_i is a single attribute, to be sorted in increasing order, i.e. $X_1 \prec ... \prec X_{s+1}$ for some ordering (say the position of the attributes in the database table corresponding to r, i.e., $X_i \prec X_j$ if X_i is o the left of X_j). The set X is a key candidate if all its subsets of size s belong to $CandNotKey(s)$, and in particular the subsets $X_1...X_{s-1}X_s$ and $X_1...X_{s-1}X_{s+1}$. Therefore, we chose to construct a set X by combining two sets Y, Z of $CandNotKey(s)$ that share a prefix of size $s-1$, such that X is the union of Y and Z and X's attributes are ordered. This is done at [3.1]: given that $Y = X_1...X_{s-1}Y_s$ and $Z = X_1...X_{s-1}Z_s$ are ordered

Algorithm 3: GenerateNextCand

% For simplicity, we denote here by s the size of keys (s corresponds to

% *size* in Algorithm 2)

$CandKey(s+1) \leftarrow \emptyset$

foreach $PrefSet$ *in* `PrefixCands`$(CandNotKey(s))$ **do**

  foreach $Y = X_1...X_{s-1}Y_s, Z = X_1...X_{s-1}Z_s \in PrefSet,\ Y_s \prec Z_s$ **do**

[3.1] $X \leftarrow X_1...X_{s-1}Y_sZ_s$

    $\text{card}(X) \leftarrow \max(\ |Z_s|.\text{card}(Y),\ |Y_s|.\text{card}(Z)\ )$

    foreach $A \in X,\ A \neq Y_s, Z_s$ **do**

[3.2] **if** $X \setminus \{A\} \in CandNotKey(s)$ **then**

[3.3] $\text{card}(X) \leftarrow \max(\ \text{card}(X),\ |A|.\text{card}(X \setminus \{A\})\ )$

      else

        $\text{card}(X) \leftarrow 0$

        exit the for loop

      end

    end

    if *card(X)* $\neq 0$ **then**

      $CandKey(s+1) \leftarrow CandKey(s+1) \cup X$

    end

  end

end

(i.e., $X_1 \prec ... \prec X_{s-1} \prec Y_s, Z_s$), and given that $Y_s \prec Z_s$, then $X = X_1...X_{s-1}Y_sZ_s$ is ordered as well by construction. Since attribute sets of size 1 are ordered, by induction, all constructed key candidates are ordered. This property is used in the function `PrefixCands`, which partitions $CandNotKey(s)$ into sets of attribute sets sharing a same prefix of size $s-1$. Since the key candidates in $CandNotKey(s)$ are ordered sets of attributes, we can sort $CandNotKey(s)$ in lexicographic order: for $X, Y \in CandNotKey(s)$, $X = X_1...X_s, Y = Y_1...Y_s$, we have $X \leq Y$ if and only if $X_i \preceq Y_i$ for $1 \leq i \leq s$. Then, we group the key candidates by prefix of length $s-1$ in one pass over the sorted set $CandNotKey(s)$. The function `PrefixCands` is also adapted from Tane [HKPT98].

When generating a key candidate X, we also compute an upper bound on its number of distinct values, stored in card(X), which is used when testing whether X is a key (see Section 3.2.1). This upper bound is based on the results of the previous level: it is the maximum among all $|A|$.card($X \setminus \{A\}$) for $A \in X$ [3.3].

3.3 Finding the foreign keys

We extract the foreign keys referencing a given key $Key = K_1...K_k$ in three steps: (1) we generate foreign key candidates, (2) we make a pruning pass over these candidates on a small set of sample rows from the database, and (3) we finally test the remaining candidates against the database.

In all the literature we are aware of, Step 3 follows directly Step 1 without any pruning.

We now describe in more detail the three steps. The steps are summarized in Algorithm 4.

Algorithm 4: ExtractForeignKeys

Input : a database d,

the key $Key = K_1...K_k$ for which we want to extract foreign keys,

ϵ_{fk}, the approximation threshold for foreign keys

Output: the foreign keys of Key

$FKset \leftarrow \emptyset$

foreach *relation* r_F *in* d **do**

  % Step 1: generate the foreign key candidates

  CollectionUINDs $\leftarrow$ `ExtractUINDs(`Key, r_F, ϵ_{fk}`)`

  FKCands(Key) $\leftarrow$ `GenerateFKCands(`Key, *CollectionUINDs*`)`

  foreach $F = F_1...F_k \in FKCands(Key)$ **do**

    % Select the number of distinct values of F

    card(F) $\leftarrow$ `SELECT COUNT(DISTINCT` $F_1, ..., F_k$`) FROM` r_F

    % Step 2: pruning pass on a sample of the database:

[4.1] **if** *passedSampleTest(* F, Key *)* **then**

      % Step 3: real test:

      card($F \bowtie Key$) $\leftarrow$ `SELECT COUNT(*) FROM`

        `(SELECT DISTINCT` $F_1, ..., F_k$

        `FROM` r_K, r_F

        `WHERE` $r_K[K_1] = r_F[F_1]$ `AND ... AND` $r_K[K_k] = r_F[F_k]$`)`

      if $card(F) - card(F \bowtie Key) \leq \epsilon_{fk}.card(F)$ **then**

        $FKset \leftarrow FKset \cup \{F \subseteq Key\}$

      end

    end

  end

end

Output $FKset$

3.3.1 Generating foreign key candidates

It is impractical to enumerate all sets of size k as potential foreign keys, since there can be $\binom{n}{k}$ of them, where n is the number of attributes. A commonly used approach to form foreign key candidates is based on previously extracted unary inclusion dependencies [KA96, KMRS92]. Another approach is described by Petit et al. [PTBK96] who tackle the more general problem of discovering inclusion dependencies. The inclusion right and left-hand side candidates are generated from sets of attributes appearing together in equijoins, within a workload of SQL queries over the database considered. Therefore this approach requires the prior knowledge of SQL queries over the database. Because we assume no prior knowledge over the database, we apply the former method, using unary inclusion dependencies. While Knobbe and Adriaans [KA96] propose determining all unary inclusion dependencies, making this a preliminary step of the general algorithm (together with the gathering of statistics), we limit ourselves to key-based UINDs. Therefore, we have to identify the keys first, so we incorporate the discovery of UINDs into the algorithm for extracting the foreign keys.

We describe the discovery of UINDs in Section 3.3.4.

Using unary inclusion dependencies to form foreign key candidates

Let's start with an intuitive example.

Example 3.3.1. Consider a database with two relations $r_1(A, B)$ and $r_2(C, D, E)$. Assume that AB is key of r_1, and that the following UINDs hold:
$C \subseteq A, D \subseteq A$, and $E \subseteq B$. Then the foreign key candidates for AB are CE and DE. But if the inclusion $E \subseteq B$ does not hold anymore, then AB has no foreign key candidate. Or if E belongs to another relation r_3, then AB has no foreign key

candidates.

Definition 3.3.1. A foreign key candidate of key $K = K_1...K_k$ is composed of k attributes $F_1...F_k$ such that

1. all F_i belong to the same relation,
2. each of them is included in the corresponding attribute of the key: $\forall i, F_i \subseteq K_i$,
3. the foreign key candidate is not equal to the key: $F_1...F_k \neq K_1...K_k$ (even though some of the F_i may be equal to some K_j)

To address the first condition, we process one relation at a time: we extract key-based UINDs from relation r, then use them to generate FK candidates, and then proceed to the next relation. Generating FK candidates is straightforward from the definition: we create all possible combinations of k attributes $F_1...F_k$ where $F_i \subseteq K_i$ and $F_1...F_k \neq K_1...K_k$. If there exists an attribute K_i such that no attribute in r references K_i, then the key has no foreign key candidate within the relation r.

3.3.2 Pruning pass over the foreign key candidates

This step is intended to eliminate quickly the candidates that are the farthest from being foreign keys. The underlying idea is that a unary inclusion dependency may hold accidentally, and only very few values of the combined left-hand side attributes are included in the values of the key.

Example 3.3.2. Let $K = K_1K_2K_3$ be a key, and $F = F_1F_2F_3$ a foreign key candidate, with the following values:

K_1	K_2	K_3	F_1	F_2	F_3
a	b	c	aa	b	c
a	bb	c	aa	bb	c
aa	b	cc	aa	bb	cc
a	b	cc			
a	bb	cc			

In this example, none of F's values are included in the set of K's values. So we can exclude F from the foreign key candidates just by looking at one of its values.

Proposition 2. *In database d, an FK candidate F is an approximate foreign key referencing key K w.r.t. error threshold ϵ_{fk} if the number of distinct values of F that are not included in K's values is at most $\epsilon_{fk}.card(F)$, where $card(F)$ is the number of distinct values of F.*

This is straightforward from the definitions of an approximate foreign key and of the error measure for FKs (see Chapter 2).

The pruning pass over foreign key candidate F is done in the function `passedSampleTest`(F, Key) ([5.1] in Algorithm 4). It is based upon Proposition 2. We extract at random a small fraction of F's distinct values, $SampleValues$. We call this fraction the pruning parameter, $PgParam$. Then, for each value v in $SampleValues$, we check whether v also appears in the set of Key's values, which we denote by $KeyValues$. This can be implemented with an SQL query:

`SELECT COUNT(*) FROM (SELECT * FROM` r_{Key} `WHERE` $Key = v$`)`

or by querying an inverted index built on the whole database. We keep a counter of the number of F's distinct values processed so far that are not included in Key's values; let's call this counter $numMisses$. Whenever v is not included in $KeyValues$, we

increment $numMisses$. As soon as $numMisses$ exceeds ϵ_{fk}.card(F), we know from Proposition 2 that F is not an approximate FK, so the function `passedSampleTest`(F, Key) returns false. For exact foreign key discovery, only one value v that is not in $KeyValues$ is enough to prune a candidate. If the bound ϵ_{fk}.card(F) is not reached during the processing of the values in $SampleValues$, then we cannot conclude whether the dependency $F \subseteq Key$ is satisfied in d based solely on the data sample, so the function returns true, and the candidate will be tested against the database.

Remark. As an important side effect, allowing approximation requires $PgParam$ to be at least equal to ϵ_{fk}.

3.3.3 Testing the remaining foreign key candidates

We test the foreign key candidates that passed the pruning step based upon Proposition 2. The number of distinct values of F that are not included in K's values is equal to the difference between the number of distinct values of F and the number of distinct values of F that are also values of Key. The former is card(F), and the latter can be computed by joining the two relations on $F = Key$. We store in card($F \bowtie Key$) the number of distinct values of the join on $F = Key$ between r_F and r_{Key}. Then, we check whether the difference card(F) $-$ card($F \bowtie Key$) is smaller than ϵ_{fk}.card(F); if it is, then F is an approximate FK for key Key in database d.

3.3.4 Discovering the unary inclusion dependencies

Discovering the unary inclusion dependencies is described in Algorithm 5.

In the following, we represent a unary inclusion dependency as a pair (LHS,RHS) where the left-hand side (LHS) is included in the right-hand side (RHS).

We are interested here in finding the left-hand sides, belonging to a relation r of the database, of all unary inclusion dependencies having a fixed right-hand side K. The brute force algorithm solving this problem consists of testing against the database all pairs (L,K) where L is an attribute of r different from K and such that L and K have the same type (e.g., text and text, or numeric and numeric). We improved on it with a simple optimization based on the statistics gathered by the first step of the general algorithm: if the number of distinct values of L is strictly greater than the number of distinct values of K, then the UIND (L,K) cannot hold. With this simple optimization, we prune the left-hand side candidates for right-hand side K.

Instead of using the attribute cardinalities, Bell and Brockhausen [BB95] extract *value restrictions*, which are upper and lower bounds on the attribute domains. They use these value restrictions to prune the candidate UINDs as follows: if $upp(L), upp(K), low(L), low(K)$ are the upper and lower bounds on the domains of attributes L, K, then (L, K) is a possible UIND if and only if $upp(L) < upp(K)$ and $low(L) > low(K)$. Value restrictions generate a more efficient pruning than simply cardinality restrictions. We only use the latter because the attribute cardinalities are available in the DBMS catalogs, while the former necessitate processing the data. The authors also exploit the transitivity of UINDs to minimize the number of accesses to the database. We found out that in practice, however, this is not a very useful optimization when extracting key-based UINDs.

Unlike Bell and Brockhausen [BB95], whose goal is to extract all unary inclusion dependencies, we are really interested in only key-based UINDs (whose right-hand sides belong to a key), since only these will be used in generating foreign key candidates. Therefore, our search space is reduced, so we did not implement all the optimizations they propose.

Moreover, our goal is to provide a general framework for key and foreign key

discovery, so we did not concentrate on all possible optimizations. However, it would be easy to integrate the ideas of Bell and Brockhausen into our algorithms: in the first step, we could include value restrictions on the statistics we gather, and then use these to prune the left-hand side candidates; and we could add after step [5.2] the following statements to exploit the UIND transitivity:

$$A_j \subseteq K_i \; and \; A_k \not\subseteq K_i \Longrightarrow A_k \not\subseteq A_j$$

$$A_j \subseteq K_i \; and \; A_j \not\subseteq A_k \Longrightarrow K_i \not\subseteq A_k$$

De Marchi et al. [MLP02] proposed an efficient way to discover all UINDs which does not use SQL queries: they generate a compact representation of the data, and from this new representation, they extract the UINDs. With their method, the restriction to key-based UINDs does not save computation. Here again, we could integrate their method into ours, but, because we concentrate on providing a general framework for key and foreign key discovery, we implemented all algorithms using SQL queries for the sake of uniformity and simplicity.

To test the inclusion of left-hand side candidate A into K, we use the same method as for foreign keys: we generate left-hand side candidates, then we perform a pruning pass over these candidates, and finally we test the remaining ones against the database. The pruning pass [5.1] is exactly the one described in Section 3.3.2 with $F = A_j$ and $Key = K_i$, which corresponds to the case where the size is equal to one. And at last, to test the inclusion of remaining left-hand side candidate A into K, we compute the number of distinct values that A has in common with K, by joining the two relations on A and K, and counting the number of distinct values of the result. The number of distinct values of the join is always less than the cardinality of A, but if the difference is smaller than $\epsilon_{fk}.card(A)$, meaning that the fraction of erroneous values is below the approximation threshold ϵ_{fk}, then the inclusion holds.

Algorithm 5: ExtractUINDs

Input : a relation r with n attributes $A_1, ..., A_n$;

the number of distinct values of each attribute, $card(A_i), i = 1...n$;

the key $Key = K_1...K_k$ for which we want to extract unary inclusion dependencies;

ϵ_{fk}, the approximation threshold for inclusions

Output: collectionUIND, the set of unary inclusion dependencies whose right-hand side belong to Key and left-hand side belongs to r

collectionUIND $\leftarrow \emptyset$

for *i = 1 ... k, when the UINDs having K_i as a right-hand side are not known* **do**

  % Extract the UINDs having K_i as a right-hand side

  CandLHS(K_i) $\leftarrow \{ A_j \mid 1 \leq j \leq n,\ A_j \neq K_i,\ |A_j| \leq |K_i|,$
  $Dom[A_j] \subseteq Dom[K_i] \}$

  foreach *$A_j \in CandLHS(K_i)$* **do**

    % Pruning pass on a sample of the database:

[5.1] **if** *passedSampleTest(A_j, K_i)* **then**

      % Real test:

      card($A_j \bowtie K_i$) $\leftarrow$ `SELECT COUNT(DISTINCT` $r.A_j$`)`
      `FROM` r, r_{K_i} `WHERE` $r.A_j = r_{K_i}.K_i$

      % If $r[A_j] \subseteq r_{K_i}[K_i]$ (almost) holds

      if $card(A_j) - card(A_j \bowtie K_i) \leq \epsilon_{fk}.card(A_j)$ **then**

[5.2] collectionUIND $\leftarrow$ collectionUIND $\cup \{r[A_j] \subseteq r_{K_i}[K_i]\}$

      end

    end

  end

end

3.4 Complexity analysis

Since all the data is kept in a DBMS, the space complexity is not an issue. Therefore, we concentrate in this section on the time complexity, which is measured in terms of database accesses.

We first examine the complexities of the database operations used in our algorithms.

- We use projection in a `SELECT DISTINCT ... FROM r` type of query. This operation requires a sort of the tuples on the attributes selected, and then a scan over all tuples, so its complexity is $O(|r| \log |r|)$. The size of the resulting set of tuples ranges from 1 (all equal) to $|r|$ (all distinct).

- We use selection in a `SELECT * FROM r WHERE ...` type of query. This operation just requires a scan over all tuples, so its complexity is $O(|r|)$.

- We use counting in a `SELECT COUNT(*) FROM r` type of query. This operation also is just a scan over all tuples, so its complexity is $O(|r|)$.

- We use equi-joins (joins whose join conditions are equalities) in a `SELECT * FROM` r_1`,`r_2 `WHERE` $r_1.X_1 = r_2.Y_1$ `AND ... AND` $r_1.X_k = r_2.Y_k$ type of query. The complexity of this operation depends on how it is executed: when executed via a nested loop, it is $O(|r_1|.|r_2|)$; when r_1 and r_2 are sorted first, it is $O(|r| \log |r|)$ where $|r| = \max(|r_1|, |r_2|)$; when an index on X or Y exists, the complexity is $O(|r| \log |r|)$ if the index is a B-tree, or $O(|r|)$ if the index is a hash table. For the rest of the complexity analysis, we assume that no index exists. The size of the resulting set of tuples ranges from 0 to $\min(|r_1|, |r_2|)$.

The algorithms often use compositions of these operations, so we adapt the complexities accordingly. As the join and projection operations are the most expensive

among the four, the complexity of compositions will be dominated by their complexities. When both operations occur, the total complexity is generally dominated by the complexity of the join operation.

Example 3.4.1. In Algorithm 4, the following query combines the three operations presented above:

SELECT COUNT(*) FROM

(SELECT DISTINCT $F_1, ..., F_k$ FROM r_K, r_F

WHERE $r_K[K_1] = r_F[F_1]$ AND ... AND $r_K[K_k] = r_F[F_k]$).

Let $m = |r_K|$, and $n = |r_F|$, and assume $m \leq n$. The complexity of the equi-join ranges from $O(n \log n)$ to $O(m.n)$. Because the size of the set of tuples returned by the equi-join is at most m, the complexity of the projection is $O(m \log m)$. Similarly, the complexity of the counting is $O(m)$. Therefore, the overall complexity is dominated by the equi-join, and is $O(n \log n)$ when the relations are sorted, or $O(m.n)$ when they are not.

We define in Table 3.1 the notation used in the rest of this section. We denote by p the maximum number of tuples over all tables, and consider that all tables have p tuples to simplify the analysis. We denote by $join(p)$ the complexity of the equi-join of two relations; $join(p) = O(p \log p)$ or $O(p^2)$ depending on the implementation of the join operation.

We now study the complexity of the key discovery module and then that of the foreign key discovery module separately.

$nAtts$	#attributes in the database
p	#tuples
$join(p)$	complexity of a join operation for two relations with at most p tuples
$PgParam$	fraction of rows involved in the pruning pass
$nKeys$	# keys found
$nKeyCands$	#key candidates
$nUINDs$	#key-based unary inclusion dependencies
$nUindCands$	#key-based UIND candidates before pruning pass
$nUindCandsPruned$	#key-based UIND candidates after pruning pass
$nFKs$	# foreign keys found
$nFKCands$	#foreign key candidates before pruning pass
$nFKCandsPruned$	#foreign key candidates after pruning pass

Table 3.1: Definition of the notations used

3.4.1 Complexity of key discovery

In the key discovery module (see Algorithm 2), we access the database to retrieve the number of distinct values for each key candidate. This corresponds to the counting operation applied to the result of a projection operation. The complexity is dominated by the projection, $O(p \log p)$ for one key candidate, and $O(nKeyCands \times p \log p)$ overall.

$$Complexity(ExtractKeys) = O(nKeyCands \times p \log p)$$

The number of key candidates is smaller in our approach than in the approaches for functional dependency discovery because of an extra pruning mechanism (line [2.1] in Algorithm 2). This mechanism, based on an estimate of the number of distinct values of a candidate, allows us to avoid some tests against the database.

3.4.2 Complexity of foreign key discovery

The main tasks of FK discovery are the following: generating FK candidates, making a pruning pass over the candidates, and testing the remaining candidates.

In the pruning pass over a FK candidate F, we first extract the number of F's distinct values, which takes $O(p \log p)$. Then, for a fraction $PgParam$ of F's distinct values, we check that they also appear in the key (i.e., the number of F's distinct values used in the pruning pass is $PgParam \times p$). The check can be done either linearly by a scan of the key values, or in $\log p$ by probing an index. We consider it to be in $O(p)$. So, the complexity of the pruning pass is quadratic in the number of tuples:

$$Complexity(Pruning) = O(nFKCands \times PgParam \times p^2)$$

We investigated experimentally the interaction between the efficiency of the pruning and the pruning parameter $PgParam$ (see next chapter).

Testing the remaining candidates is done through the query of Example 3.4.1 (see page 55), whose complexity is $O(join(p))$.

$$Complexity(TestFKCands) = O(nFKCandsPruned \times join(p))$$

Finally, generating foreign key candidates requires determining key-based unary inclusion dependencies (see Algorithm 5). The process for UINDs is similar to that for general FKs, and so is the complexity.

$$\begin{aligned} Complexity(ExtractUINDs) = O(&nUindsCands \times PgParam \times p^2 \\ &+ nUindsCandPruned \times join(p)) \end{aligned}$$

The overall complexity is the sum of the three equations above.

Influence of data characteristics on the complexity

The data characteristics influencing the complexities vary greatly from one database to another. The complexities also depend on input parameters: error thresholds and number of values involved in the pruning pass, which we discuss in next section.

We saw in Section 2.1.5 that the numbers of keys and key candidates, $nKeys$ and $nKeyCands$, can be exponential in the number of attributes. In addition to the fact that interesting keys tend to be small in practice, we chose to limit the maximal key size also to avoid this exponential explosion.

The key-based unary inclusion dependencies depend on the number of attributes included in keys since these are the right-hand sides of the UINDs. While in practice the right-hand sides form a small subset of all attributes, in the worst case, the union of the keys covers all attributes in the database, so that there are $nAtts$ right-hand sides. Also, even though generally domains are of different types (e.g., integers,

strings, dates), and therefore are disjoint, it may occur that all of them are of the same type, and that they are all included in the domains of key attributes. Then, there are $nAtts$ left-hand sides per right-hand side. So, in the worst case, $nUinds = nUindCands = nUindCandsPruned = nAtts^2$.

The FK candidates are generated from key-based UINDs, so they depend on the number of keys and key-based UINDs. We saw that in the worst case, there can be a great number of them.

Influence of input parameters on the complexity

The pruning pass over the foreign key candidates is useful only when the difference between the number of candidates before and after pruning justifies the additional cost of the pruning step.

Without the pruning pass, the complexity of testing the foreign key candidates is:

$$O(\ (nUindCands\ +\ nFKCands) \times join(p)\ )$$

With it, the complexity becomes:

$$O(\ (nUindCandsPruned\ +\ nFKCandsPruned) \times join(p)$$
$$+\ (nUindsCands\ +\ nFKCands) \times PgParam \times p^2\ )$$

So the pruning step is worthwhile only when:

$$(nUindsCands\ +\ nFKCands)\ \times\ PgParam\ \times\ p^2\ \ll$$
$$(nUindCands\ -\ nUindCandsPruned\ +$$
$$nFKCands - nFKCandsPruned)\ \times\ join(p)$$

When the fraction $PgParam$ of tuples involved in the pruning pass is small, the pruning step should be less efficient than when $PgParam$ is large. On the other hand, as $PgParam$ increases, the cost of the pruning pass grows, so the number of

candidates after pruning has to be significantly smaller than the number of initial candidates for the pruning step to be worthwhile.

The intuitive idea is that the pruning step should only be used to discard the candidates that are very far from being foreign keys, e.g., which have just a couple of tuples in common with the key. These candidates do not require testing on many rows, so a small $PgParam$ is enough to eliminate them, and the pruning step is inexpensive. For the remaining candidates, we use the full test. Indeed, if we wanted to discard them during the pruning pass, we would have to process many tuples because they share many values with the key, so we would need a large $PgParam$ and the pruning step might become too costly relative to its efficacy. Unfortunately, the tradeoff between the cost of the pruning step and its performance depends on the dataset.

The other input parameters (other than $PgParam$) are the error thresholds for approximate keys and FKs. Non-zero thresholds increase the number of key-based UINDs, keys, FKs and all candidates.

When the approximation threshold ϵ_{fk} for the foreign keys increases, the fraction $PgParam$ of tuples involved in the pruning pass has to increase as well since we must have $PgParam \geq \epsilon_{fk}$ (see Section 3.3.2). Hence, for high thresholds, the pruning step may be too expensive relative to its benefit, and may have to be skipped. But such high thresholds would probably give coarse, not very interesting results.

We investigate in the next chapter the effect of the input parameters on the performance of the algorithm through experimentation.

Chapter 4

Experiments

4.1 Goals of the tests

- We evaluate the gain in terms of database accesses of extracting only key-based UINDs versus all UINDs.

- We evaluate the efficiency of the pruning pass, both on synthetic and real-life datasets with various amount of noise.

  We examine the general behavior of the pruning pass efficiency in relation to the pruning parameter $PgParam$. In particular, we study how to set $PgParam$ to minimize the running time of the algorithm.

  We then analyze the influence of the dependency size on the pruning pass, that is, whether the pruning pass behaves differently on unary and composite dependencies. In addition, we investigate the influence of the approximation threshold for FKs, ϵ_{fk}.

  Finally, we verify experimentally that the pruning time is quadratically dependent on the data size (i.e., the number of tuples).

4.2 Experimental setup

All the datasets used in the experiments were stored in IBM DB2 universal database Version 6.1, for Solaris. We implemented the algorithms described in Chapter 3 in C, with embedded SQL to access the databases. Because of the randomness in our algorithm (within the pruning pass), we ran it 5 times on the whole set of databases, and we took the average over the 5 runs for each dataset.

First of all, we define the metrics used to evaluate the effectiveness of the pruning pass. Then, we present the datasets used in the experiments. Finally, we describe the setting of the input parameters of the algorithm (such as the approximation thresholds for keys and FKs).

4.2.1 Metrics used to evaluate the impact of the pruning pass

Recall that our approach for identifying the foreign keys of a given key consists of three steps:

1. generating FK candidates
   - for FKs of size 1, these are the key-based UIND candidates;
   - for FKs of size > 1, these are generated from key-based UINDs.
2. pruning the FK candidates
3. testing the remaining FK candidates.

The pruning pass (Step 2) aims at minimizing the number of FK candidates involved in Step 3, because this last step is the most expensive. Thus, there is a tradeoff between the extra cost of the pruning pass and the savings it provides. The pruning

pass is useful only when the savings exceed the cost.

We present now the metrics used to evaluate whether the pruning pass is worthwhile, and how efficient it is. These metrics are based upon the following statistics, recorded during the execution of the algorithm for each arity (i.e., number of attributes of the key/FK):

- $nFKCands$, the number of FK candidates. In each pass, this is recorded after Step 1. Note that FK candidates of size 1 are far more numerous than for other sizes because the generating process is different.
- $nFKCandsPd$, the number of FK candidates after the pruning of Step 2.
- $nFKs$, the number of FKs. This is the number of candidates that passed the test of Step 3.

In addition, we gather statistics about the whole set of key-based UINDs (not only those associated with keys of size 1).

Degree of pruning. We measure how effective the pruning pass is in reducing the number of FK candidates involved in the full test of Step 3 to the set of actual FKs. For instance, assume that there are 20 FKs holding in a database instance, and that our algorithm finds 50 candidates (i.e., $nFKs = 20$ and $nFKCands = 50$). Then the pruning pass is most effective when it prunes all the 30 candidates that are not actual FKs (i.e., when $nFKCandsPd = 20$).

We define the degree of pruning as follows:

$$DegPg = \frac{nFKCands - nFKCandsPd}{nFKCands - nFKs}$$

When $nFKCands = nFKs$, the pruning pass is useless, and does not result in any improvement over the algorithm without the pruning pass, therefore we chose to define $DegPg = 0$ in this case. The degree of pruning lies in the interval $[0, 1]$, where 1 corresponds to a perfectly effective pruning, and 0 to a totally ineffective pruning.

We chose to use this measure rather than the number of candidates pruned over the number of candidates:

$$DegPg_2 = \frac{nFKCands - nFKCandsPd}{nFKCands}$$

because $DegPg_2$ reflects only the quantity of pruning done, without taking into account its quality (i.e., how close the candidates after pruning are of the final FKs). Indeed, in our previous example, the pruning is most effective, but $DegPg_2$ is only 0.6, while $DegPg = 1$ expresses the intuitive idea that the pruning was perfect.

The degree of pruning measures the *decrease*, due to pruning, of the number of candidates tested against the database that are not FKs. Consider the case where $nFKCands = 30, nFKs = 27$, and $nFKCandsPd = 29$. The degree of pruning is low: 0.33. However, we only test two superfluous candidate, which is rather good. Thus, we introduce another measure to compute the number of excess tests against the database.

Pruning ratio and FKcand ratio. We measure how the number of candidates tested against the database in Step 3 compares to the number of actual FKs. To evaluate the usefulness of the pruning pass, we record this ratio both when there is a pruning pass (we call it pruning ratio), and when there is not (we call it FKcand

ratio), i.e., with and without Step 2 of the algorithm.

$$PgRatio = \frac{nFKs}{nFKCandsPd}$$
$$FKcandRatio = \frac{nFKs}{nFKCands}$$

Here again, both ratios lie in the interval $[0, 1]$, where 1 corresponds to no excess test against the database, while 0 corresponds to the case where there are no FKs, but some candidates are tested.

While these measures express how good the results after pruning are, they do not account for the cost necessary to obtain them: the extra time needed to execute the pruning pass. Thus, to know whether the pruning pass was worthwhile, we introduce a last measure, called time ratio.

Time ratio. We measure the running time of the algorithm without the pruning pass, $TimeNoPg$, and with it, $TimeWithPg$. Then

$$TimeRatio = \frac{TimeWithPg}{TimeNoPg}$$

When $TimeRatio < 1$, the pruning pass was worth it, otherwise, it is faster to use the version without pruning.

All the measures defined above depend on the fraction $PgParam$ of distinct tuples involved in the pruning pass: the larger $PgParam$ is, the more effective the pruning will be, but the more expensive the pruning pass is. Therefore, there is a tradeoff between the gains and the cost. In the next section, we study the influence of $PgParam$ on the different measures, and discuss what value is the best choice for $PgParam$.

4.2.2 Datasets used in the experiments

The goal of the algorithm presented in this thesis is to determine the data model (keys and FKs) which best fits the database considered. More precisely, we can view a database as the sum of a clean database and some noise, and the model to discover corresponds to that of the clean database. In our experiments, we used a few real-life nearly noiseless databases whose data models are known, and we added various amounts of noise to them to simulate noisy databases. We used the given data models to check whether those output by our algorithm are correct.

So our noisy databases are all synthetically generated. But as there is no general model of noise in databases (i.e., the characteristics of the noise in two databases can be totally different), we consider that our synthetic noisy data are as appropriate for the experiments as real-life noisy data would be. Moreover, using synthetic noisy data allows us to control the amount of noise in our experiments.

Our real-life clean databases are of medium size w.r.t. both their schema and their number of tuples. They were all carefully designed, thus most of the foreign keys are of size 1 or 2, which is the case of most databases in practice. We generated synthetic data with different characteristics (i.e., different key and foreign key sizes), to test our algorithm on other configurations.

Real-life datasets. These databases are all publicly available on the Internet.

- Amalgam is a bibliographic database, with 101 attributes distributed in 15 tables, and up to 10000 tuples per table. We used Schema 1 defined by Miller et al. [MFH+01].
- Genex [Gen] is a repository of gene expression data, with 224 attributes distributed in 30 tables, and up to 150000 tuples per table.

- Janus Core Log (which we call simply Janus in the following) [Oce] is part of the Janus database, containing 450 tables of Ocean Drilling Program's marine geoscience data. The Core Log subset of the Janus database is composed of 176 attributes distributed in 22 tables, and the sample we used contained up to 9000 tuples per table.

- Mondial [May99] is a geographical database, with 102 attributes in 28 tables, and up to 8000 tuples per table.

Synthetic data. The 16 synthetic datasets are small w.r.t. both their schema and their number of rows: around 30 attributes in 3 tables, and 5000 rows. They were generated to test worst cases of the algorithms: the domains all have the same type and many common values, thus overlapping much more than in the real-life datasets we used. Besides, the synthetic datasets present different characteristics (i.e., different key and foreign key sizes).

We used constraint programming [Lab] to generate data respecting conditions on the number of keys, their sizes, their number of FKs with the size of these FKs. The classical synthetic data generators (e.g., those for classification, association rules) are not suitable for our purpose. The best tool we are aware of is Toxgene [BMKL02]; but we did not use it because it requires knowing the data model in advance, while our tool constructs one which fits our conditions. However, unlike Toxgene, our tool is not scalable, which explains why our synthetic datasets have small schemas. This is due to the fact that the problem of generating a database satisfying a set of integrity constraints specified by a user is a very difficult one. Bry et al. explain that the set of integrity constraints can be expressed by a set of first-order logical formulas, and the goal is to find a finite model (because a database is finite) for this set of first-order logical formulas [BEST98]. But the question of whether a set of first-order

logical formulas is finitely satisfiable is semi-decidable[1], and the question of whether a set of first-order logical formulas is unsatisfiable is semi-decidable, so the question whether a set of first-order logical formulas is satisfiable though not finitely is not even semi-decidable.

Noisy data. We describe here how we generated approximate data from our clean datasets. Given approximation (or noise) degree $noiseDeg$ as a percentage, we proceed as follows:

- for each table, we select at random $noiseDeg$ percent of the tuples $t_1, ..., t_k$;
- for each $t_i = (v_1, ..., v_n)$: either duplicate t_i with probability 20%, or modify some values in t_i with probability 80%. When modifying t_i, each value is replaced with probability 50% by a random value having the same type, and with probability 10% by a NULL value (so a value is not modified in 40% of cases). Duplicating tuples adds noise to keys only, not to FKs, while modifying some values in tuples adds noise to both keys and FKs.

We created various noisy versions of our datasets: we used $noiseDeg = 0.5$ and 1%.

4.2.3 Setting of the algorithm's input parameters

We study which values of ϵ_{key} and ϵ_{fk}, the input approximation thresholds for keys and FKs respectively, give a good enough model for all our datasets. Of course, the threshold between good enough and unacceptable models is subjective, and can vary according to the applications. In this thesis, we consider that a model is good enough

[1] A language L is semi-decidable if there exists a Turing machine M that accepts every string in L and does not accept any string not in L. (M either rejects or goes in an infinite loop.)

when few dependencies are missing, and if possible, when those missing dependencies are of minor importance in the original model. For instance, we consider that it is acceptable to miss the key of a small table which is not referenced; however, dependencies expressing relationships between tables, such as FKs and referenced keys, are of major importance. Because we do not know in advance how noisy the database is, we want to find a pair $(\epsilon_{key}, \epsilon_{fk})$ that works well on most databases, clean as well as noisy, and then we will use these values as input parameters for our experiments.

For this experiment, we chose two of our real-life datasets: Mondial because its data model has the characteristics of most well-designed databases, with keys of size 1 and 2; and Janus because its data model has large composite keys, of size 3, 4 and 5. However, the large composite keys in Janus' data model are actually superkeys in Janus' data, so we replaced them in Janus' data model by their smallest subset that is key in the data (if there are several such subsets, we choose one at random). For instance, the primary key of the table "Core" in Janus' original data model is {Leg, Site, Hole, Core, Core_type}, but {Site, Core, Core_type} is key in Janus' data, so we used the latter in Janus' updated data model. We summarize in Table 4.1 the keys and foreign keys of both Mondial and Janus data models.

	Number of keys		Number of FKs	
Database	total	for size 1/2/3/4	total	for size 1/2/3/4
Mondial	25	10/15/0/0	48	34/14/0/0
Janus	22	17/4/1/0	36	30/4/2/0

Table 4.1: Summary of keys and foreign keys in Mondial and Janus data models

Finding ϵ_{key}. We tested our algorithm on each original dataset (corresponding to $noiseDeg = 0$), as well as on the noisy versions of it. We fixed the input parameter

ϵ_{fk} to 0 since FKs are of no concern in this particular study, and we experimented with various values for ϵ_{key}. Then we compared the keys output by our algorithm to those of the data model describing the real-life dataset. We recorded in Table 4.2 the number of keys present in the data model that are missed by our algorithm.

		# missing keys for	
$noiseDeg$	ϵ_{key}	Mondial	Janus
0	0	0	0
0.005	0	4	3
	0.005	0	0
	0.01	0	0
0.01	0	6	3
	0.005	2	0
	0.01	0	0

Table 4.2: Number of keys present in the model but not found by our algorithm

We can see in Table 4.2 that as ϵ_{key} increases, the number of keys missed by our algorithm decreases. We found that $\epsilon_{key} = 0.005$ is a good choice: the number of missing keys decreases to 0 for all noise degrees but $noiseDeg = 0.01$ on the Mondial database, and in this case, the two missing keys are of minor importance in the original model (they are not referenced). Moreover, as we consider datasets with small noise degrees (up to 1%), this value should cover most of the noise on keys.

Finding ϵ_{fk}. We repeat the previous experiment, except that this time, the parameter varied is ϵ_{fk}, and ϵ_{key} is fixed to the value we just chose: 0.005. We construct Table 4.3 similarly to Table 4.2, the table for finding ϵ_{key}.

Our original Janus data is not perfectly clean: one FK is missing when $noiseDeg =$

		# missing FKs for	
$noiseDeg$	ϵ_{fk}	Mondial	Janus
0	0	0	1 (1/0/0/0)
	0.1	0	1 (1/0/0/0)
0.005	0	20 (11/9/0/0)	13 (10/2/1/0)
	0.01	16 (10/6/0/0)	12 (9/2/1/0)
	0.02	9 (7/2/0/0)	12 (9/2/1/0)
	0.05	7 (6/1/0/0)	12 (9/2/1/0)
	0.1	6 (5 /1/0/0)	12 (9/2/1/0)
0.01	0	34 (21/13/0/0)	13 (10/2/1/0)
	0.01	29 (17/12/0/0)	13 (10/2/1/0)
	0.02	16 (12/4/0/0)	13 (10/2/1/0)
	0.05	9 (8/1/0/0)	12 (9/2/1/0)
	0.1	8 (7/1/0/0)	12 (9/2/1/0)

Table 4.3: Number of FKs present in the model but not found by our algorithm, with $\epsilon_{key} = 0.005$. The details per size are inside the parentheses.

0 and $\epsilon_{fk} = 0$. This FK has only a few distinct values, including a noisy one, so a large ϵ_{fk} (larger than 0.1) is needed to recover the FK. No value for ϵ_{fk} allows us to recover all missing FKs. We chose $\epsilon_{fk} = 0.02$ because it is a small value for which the number of missing FKs has decreased significantly.

We found that $\epsilon_{key} = 0.005$ and $\epsilon_{fk} = 0.02$ did work reasonably well on all of our datasets, clean as well as noisy, so we use these values as input parameters in the rest of our experiments. Even though these values allow us to recover most of the data models for our real-life datasets whatever the noise degree, there may be other databases for which these values are not suitable. We feel that in general, they should allow us to discover most of the keys and foreign keys of the data model, provided that the data is not very noisy. For very dirty data, no technique is capable of finding the right dependencies.

Finally, the last input parameter of our algorithm is the pruning parameter: the fraction $PgParam$ of tuples involved in the pruning pass. We will study how to best set $PgParam$ in Section 4.4.

4.3 Key-based unary inclusion dependencies

We evaluate in this section the gain, in terms of database accesses, of extracting only key-based UINDs instead of all UINDs.

For each of our real-life databases (with no noise added), we record in Table 4.4 the number of UIND candidates both when we extract all UINDs and when we extract only key-based UINDs, and the percentage of attributes that are included in a key. We also express the number of key-based UIND candidates as a percentage of all

UIND candidates. For this experiment, the stopping criterion used in the algorithm is the limited size criterion (i.e., we search all integrity constraints up to size 4).

Database	#UINDCands	#KeyBasedUINDCands (as % of UINDCands)	Percentage of attributes included in a key
Amalgam	2533	1309 (i.e., 52%)	48%
Genex	14377	8712 (i.e., 60%)	46%
Mondial	3175	2257 (i.e., 71%)	69%
Janus	8753	4599 (i.e., 52%)	42%

Table 4.4: Extracting all UINDs versus only key-based UINDs

We found that, on these four datasets, 52% to 71% of UIND candidates are key-based. Hence, limiting ourselves to key-based UINDs saves between one third and one half in database queries. The raw numbers show that these savings are significant: there are 1,000 to 6,000 fewer key-based UIND candidates than UIND candidates. As each candidate requires a database query in order to check whether the UIND actually holds in the database, we avoid about 1,000 to 6,000 queries. The percentage of attributes included in keys is slightly smaller than the percentage of UIND candidates that are key-based, due to our candidate generation process. Indeed, an attribute L is a left-hand side candidate for a given attribute R provided that L and R have the same type, and that $|L| \leq |R|$, where $|L|$ is the number of distinct values of L. Because attributes in keys have more distinct values than attributes not in keys, there are more UIND candidates with a key attribute as the right-hand side.

4.4 Efficiency of the pruning pass

First of all, we show in Table 4.5 that the pruning pass can be very effective in some cases. We recorded the running time of our algorithm both with and without the pruning pass, on our four real-life datasets (without noise added). We computed the fraction of running time without pruning that we avoided by performing a pruning pass; this fraction measures the benefit of the pruning pass. For this experiment, we used the limited size stopping criterion, $\epsilon_{key} = \epsilon_{fk} = 0$, and various values for the fraction $PgParam$ of distinct values involved in the pruning pass (the best value for each dataset).

Database	$PgParam$	Running time without pruning	Running time with pruning	Gain due to the pruning pass (in % of running time without pruning)
Amalgam	0.01%	6min21s	5min18s	17%
Mondial	0.1%	2min32s	1min6s	57%
Genex	0.1%	21min25s	16min8s	25%
Janus	1%	55min42s	16min31s	70%

Table 4.5: Running times of our algorithm with and without pruning pass on original data, with $\epsilon_{key} = \epsilon_{fk} = 0$.

From Table 4.5, we see that the pruning pass allows us to save 17 to 70% in running time, so it can be very effective. This justifies the more detailed study that follows, where we examine the effectiveness of the pruning pass, and which parameters influence it.

4.4.1 General behavior of the pruning pass in relation to the pruning parameter

We study in this section the general behavior of the pruning pass, and how this behavior depends on the pruning parameter $PgParam$.

For a general analysis, we experimented on most of the datasets described in Section 4.2.2. We ran our algorithm on our real-life datasets (Mondial, Genex, Janus) and synthetic data, as well as on noisy versions of Mondial and Janus. We used the input approximation thresholds of Section 4.2.3: $\epsilon_{key} = 0.005$ and $\epsilon_{fk} = 0.02$.

First of all, we examine whether the pruning pass is worthwhile, and for which value of the pruning parameter the pruning is most effective. As the pruning pass requires that $PgParam > \epsilon_{fk}$ and ϵ_{fk} is set to 0.02, we experimented with $PgParam$-values ranging from 0.025 to 0.10. Figure 4.1 shows the ratio between the running time with pruning and without pruning, versus the pruning parameter. All these graphs have the same shape: the time ratio increases as the pruning parameter grows. However, their positions with respect to the line of equation $y = 1$ differ according to the data: on synthetic data, the time ratio is bigger than 1 for all tested values of $PgParam$, while on Janus, the time ratio is smaller than 1 for these same values. Finally on Mondial, the time ratio crosses the line $y = 1$ around $PgParam = 0.04$.

So, the pruning pass (PP) is most effective for smaller value of the pruning parameter (0.025 in our experiments). But this optimal value of $PgParam$ can correspond to a worthwhile or detrimental pruning pass, depending on the dataset. (Note that a detrimental PP is such that $TimeWithPg > TimeNoPg$.)

We now investigate why the time ratio increases with $PgParam$, and why the

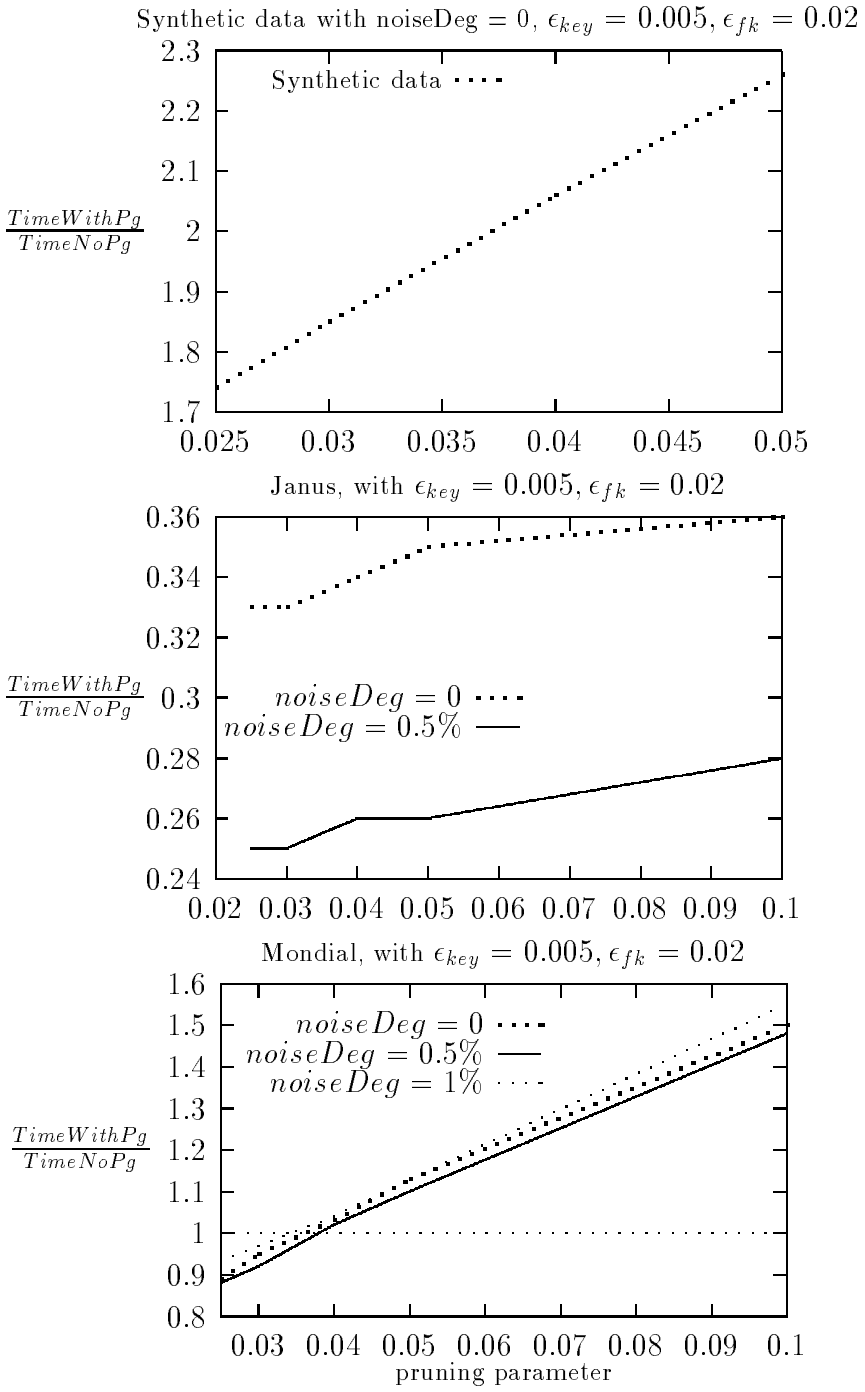

Figure 4.1: Time savings

position of the curve w.r.t. the line of equation $y = 1$ changes according to the data. We measure the performance of the pruning over all FK candidates with the metrics described in Section 4.2.1, namely, (1) the FK candidate and pruning ratios, $FKcandRatio$ and $PgRatio$, which measure the number of superfluous candidate tests against the database, without and with pruning respectively (the closer to 1, the fewer superfluous tests there are); and (2) the degree of pruning, $DegPg$, which measures how many candidates have been discarded by the pruning pass. For conciseness, we limited our investigation to one version of each real-life database: Mondial and Janus with $noiseDeg = 0.5\%$, and Genex with $noiseDeg = 0$, but the results are similar for other noise degrees. We present the results in Table 4.6.

Without pruning pass, the algorithm performs many unnecessary database queries, as shown by very low FK candidate ratios: from 0.04 for Janus to 0.16 for synthetic data. Recall that $FKcandRatio = \frac{\#FKs}{\#candidates}$, and that each candidate is tested against the database via a query. In the ideal case, the algorithm tests only the candidates for which the inclusion holds, which corresponds to the number of candidates being equal to the number of FKs and $FKcandRatio = 1$. When there are more candidates than FKs (i.e., when $FKcandRatio < 1$), superfluous database queries are performed.

Introducing a small amount of pruning ($PgParam = 2.5\%$) reduces greatly this number of unnecessary database queries on Mondial, Janus and Genex: the pruning ratios are more than 6 times bigger than the FK candidate ratios. Hence, the pruning is very effective on these datasets, as shown by degrees of pruning greater than 0.94. But increasing the pruning parameter brings little improvement to the effectiveness of the pruning pass, as demonstrated by slowly increasing degrees of pruning and pruning ratios on Mondial, Janus, Genex. Even when the performance of the PP remains relatively steady while $PgParam$ grows, the cost of performing the pruning

Database	Efficiency measure	Pruning parameter (in %)				
		2.5	3	4	5	10
Mondial (noiseDeg = 0.5%)	FKcandRatio	0.09				
	PgRatio	0.62	0.63	0.67	0.70	0.77
	DegPg	0.94	0.94	0.95	0.96	0.97
Genex (noiseDeg = 0)	FKcandRatio	0.07				
	PgRatio	0.69	0.69	0.69	-	-
	DegPg	0.96	0.96	0.96	-	-
Janus (noiseDeg = 0.5%)	FKcandRatio	0.04				
	PgRatio	0.59	0.61	0.61	0.61	0.62
	DegPg	0.97	0.97	0.97	0.97	0.97
Synthetic data (noiseDeg = 0)	FKcandRatio	0.16				
	PgRatio	0.18	0.19	0.22	0.26	0.53
	DegPg	0.12	0.18	0.32	0.45	0.81

Table 4.6: Efficiency of the pruning pass on all candidates, with input parameters $\epsilon_{key} = 0.005$ and $\epsilon_{fk} = 0.02$

still increases, so the cost of the PP increases faster than the benefits it generates. This explains why the gain from pruning decreases with $PgParam$.

The behavior of the PP is slightly different on synthetic data. Indeed, the pruning pass is not as effective for small $PgParam$: the pruning ratio is nearly equal to the FK candidate ratio for $PgParam = 2.5\%$, and the degree of pruning is small, at only 0.12 as opposed to more than 0.94 on real-life data.

Overall, the best compromise benefit/cost is reached for the smallest $PgParam$, 2.5%, and from there, the effectiveness of the PP decreases on all datasets because the cost of pruning grows faster than its benefits. For real-life datasets, pruning with $PgParam = 2.5\%$ is relatively inexpensive, and it greatly reduces the number of superfluous database queries, so the optimal value of the pruning parameter corresponds to a worthwhile PP. On synthetic data however, the optimal value $PgParam = 2.5\%$ corresponds to a detrimental PP, where the cost always surpasses the benefits.

We now study why, on real-life data, the PP is so effective for a small $PgParam$, and why its performance increases very little for larger $PgParam$-values. The candidates can be classified in two groups: the "accidental candidates", that have very few values in common with the right-hand side, and the "expected candidates", that share more than a few values with the right-hand side. To discard an accidental candidate, we have to process only a couple more tuples than required by ϵ_{fk} (i.e., a very small pruning parameter suffices). On the other hand, to be able to discard some of the expected candidates, we need to examine many more tuples, that is, we need a large $PgParam$.

We investigated how many values are shared by the accidental and expected candidates and their right-hand side (RHS). We chose only one real-life dataset, Mondial, and one synthetic dataset for this experiment. Mondial is representative of real-life

datasets, and the synthetic database corresponds to a special case, where all attributes share several values. We computed the average percentage of common values between candidates and RHSs, for two $PgParam$-values: the smallest value used in previous tests, 2.5%, and a larger one, 5%. The candidates that are discarded in the PP correspond to the accidental candidates, and the others are the expected candidates. We present the results in Table 4.7.

Database	Pruning parameter (in %)	Avg percent. of common values between rhs and	
		pruned cand. lhs	not pruned cand. lhs
Mondial	2.5	1.47	86.38
	5	2.25	96.08
Synthetic data	2.5	43.95	69.80
	5	45.18	79.98

Table 4.7: Average percentage of common values between the candidate left-hand sides and their right-hand sides, on databases with noiseDeg = 0

On Mondial, when $PgParam = 2.5\%$, the accidental candidates share only 1.47% of their values with their RHSs; while the expected candidates share as much as 86.38% with their RHSs.

Recall that candidates are either accidental or expected. As accidental and expected candidates are defined in relation to the PP, these two sets vary with the pruning parameter. When $PgParam$ increases, some of the candidates that were before expected candidates become accidental, thus the average percentage of common values between accidental candidates and their RHSs increases, as can be seen in Table 4.7.

The high average of common values between accidental keys and their RHSs

on synthetic data explains why the performance of the PP is so poor with small $PgParam$, and why it increases slowly with $PgParam$. The percentage of common values between accidental keys and their RHSs are much larger for synthetic data than for Mondial: 45% vs. 2%. This explains why the performance of the PP is lower on synthetic data than on real-life data: the pruning ratio is at most 0.26 on synthetic data while it reaches 0.61 on Janus, and 0.70 on Mondial (see Table 4.6).

These results confirm the observations made in Section 3.4 (Complexity section), namely that the PP is heavily influenced by the degree of overlap in the different domains. We generated synthetic data to test our algorithm on worst case data, so if the number of common values between candidates and RHS is high on synthetic data, it is on purpose. The experiments corroborate that overlapping domains represent indeed a bad case for our algorithm. Indeed, the time ratio is always smaller than 1 for synthetic data, so it is better not to perform a PP on these data.

Overall, we saw that the best value for the pruning parameter is very close to the FK approximation threshold ϵ_{fk}, for instance $PgParam = 2.5\% = 0.025$ when $\epsilon_{fk} = 0.02$. But even for the best value of the pruning parameter, the PP may be detrimental, and one should avoid performing it in some cases. This happens in particular when the different attribute domains are overlapping too much.

4.4.2 Influence of dependency size

We study in this section the influence of dependency size on the effectiveness of the pruning pass. We used the same set of experiments as in Section 4.4.1, except that here, we detail the performance results per dependency size. We found that the pruning pass has the same effects for each of the sizes larger than one, hence we distinguish

only between unary and composite dependencies in this section. The results are presented in Table 4.8 for key-based UINDs, and Table 4.9 for composite dependencies.

Database	Efficiency measure	Pruning parameter (in %)				
		2.5	3	4	5	10
Mondial (noiseDeg = 0.5%)	FKcandRatio	0.09				
	PgRatio	0.60	0.61	0.65	0.69	0.76
	DegPg	0.94	0.94	0.95	0.96	0.97
Genex (noiseDeg = 0)	FKcandRatio	0.07				
	PgRatio	0.69	0.69	0.69	-	-
	DegPg	0.97	0.97	0.97	-	-
Janus (noiseDeg = 0.5%)	FKcandRatio	0.04				
	PgRatio	0.56	0.59	0.59	0.59	0.60
	DegPg	0.97	0.97	0.97	0.97	0.97
Synthetic data (noiseDeg = 0)	FKcandRatio	0.11				
	PgRatio	0.12	0.13	0.15	0.18	0.42
	DegPg	0.11	0.18	0.31	0.45	0.80

Table 4.8: Efficiency of the pruning pass on key-based UINDs, with input parameters $\epsilon_{key} = 0.005$ and $\epsilon_{fk} = 0.02$

First of all, notice that the figures in Table 4.8 are nearly the same as in Table 4.6, so the general behavior of the pruning pass is actually shaped by its behavior on unary candidates. Because there are far more unary candidates than composite candidates, the effectiveness of the PP is dominated by how it fares on unary candidates.

For most unary candidates, the inclusion dependency does not hold in the database, as shown by the very low values of the FK candidate ratio: from 0.04 for Janus to

Database	Efficiency measure	Pruning parameter (in %)				
		2.5	3	4	5	10
Mondial (noiseDeg = 0.5%)	FKcandRatio	1				
	PgRatio	1	1	1	1	1
	DegPg	0	0	0	0	0
Genex (noiseDeg = 0)	FKcandRatio	0.68				
	PgRatio	0.74	0.74	0.74	-	-
	DegPg	0.25	0.25	0.25	-	-
Janus (noiseDeg = 0.5%)	FKcandRatio	0.46				
	PgRatio	0.70	0.70	0.70	0.70	0.70
	DegPg	0.65	0.65	0.65	0.65	0.65
Synthetic data (noiseDeg = 0)	FKcandRatio	0.78				
	PgRatio	0.79	0.81	0.85	0.89	0.98
	DegPg	0.05	0.10	0.22	035	0.54

Table 4.9: Efficiency of the pruning pass on composite FK candidates, with input parameters $\epsilon_{key} = 0.005$ and $\epsilon_{fk} = 0.02$

0.09 for Mondial (see Table 4.8). This is due to the fact that our candidate generating process is not very constraining: an attribute L is a left-hand side candidate for a RHS K provided that L and K have the same type, and L has fewer distinct values than K. Thus, an attribute L can be a candidate even if it has no value in common with the right-hand side K. So, there is a huge number of left-hand side candidates for each right-hand side, and most of them are accidental candidates.

The pruning pass is very effective on unary candidates: the degree of pruning is above 0.94 for Mondial, Genex and Janus, so a large portion of the candidates are discarded during the pruning pass. However, there still remain candidates for which the inclusion dependency does not hold in the database: the pruning ratio lies between 0.56 and 0.69 for Mondial, Genex and Janus.

Unlike for key-based UINDs, the pruning pass discards few composite candidates: for Mondial and Genex, the pruning ratio is equal or nearly equal to the FKcand ratio, and for Janus, $PgRatio \approx 1.5 \times FKcandRatio$ (see Table 4.9). And the FKcand ratio is high: from 0.46 for Janus to 1 for Mondial, so even without pruning pass, there are not many superfluous tests against the database. These differences between the composite and unary cases come from the fact that most unary candidates are accidental, while most composite candidates are expected. Indeed, a composite FK candidate $F_1...F_k$ for key $K_1...K_k$ is such that $F_i \subseteq K_i$ for all $1 \leq i \leq k$, so the candidate is likely to share many values with the key. Therefore, the probability of pruning a composite candidate is very low. We compute this probability on an example taken from Janus: a composite candidate of size 2 has 418 distinct values, including 399 values shared with its key (i.e., 95% of the candidate's distinct values are shared with its key). To discard this candidate, we need to pick at least $\lceil \epsilon_{fk}.418 \rceil = 9$ values that are not common with the key during the pruning pass. Assume that $PgParam = 10\%$. We can use up to $\lceil PgParam.418 \rceil = 21$ values in the pruning

pass. So the probability of discarding the candidate is:

$$\begin{aligned} ProbaPg &= P\{\text{pick at least 9 values not common with the key among 21 values}\} \\ &= \sum_{k=9}^{21} \binom{42}{k} 0.05^k \, 0.95^{42-k} \\ &= 1.93 \times 10^{-4} \end{aligned}$$

Note that this probability increases up to 0.88 when $\epsilon_{fk} = 0$, so the pruning pass can be effective on composite candidate when ϵ_{fk} is very small. The composite candidates discarded during the pruning pass are accidental candidates, which share few values with their key. These accidental candidates often belong to the same table as the key, and also have some attributes in common with the key. For instance, $F = Firstname, Gender, Lastname$ can be a FK candidate for the key $K = Firstname, Mid-init., Lastname$, but F and K have probably very few common values.

The effectiveness of the pruning pass on composite candidates remains constant when the pruning parameter increases (see Table 4.9). Indeed, the few accidental candidates are discarded with a very small $PgParam$, but the expected candidates share so many values with their keys that a very large $PgParam$ is required to prune them. Therefore, if the pruning pass is to be performed on composite candidates, it is best to use very small $PgParam$-values because the cost of pruning is minimized without a loss of performance. However, as the benefit due to pruning is low, performing a PP may be more costly than its gains.

We saw that the general effectiveness of the PP, as studied earlier, is actually shaped by the effectiveness on key-based unary candidates, because a large part of the total number of candidates are unary. So for unary candidates, it is most effective to use a small $PgParam$ value, close to ϵ_{fk}, because it brings a good pruning effectiveness

at low cost. For instance when $\epsilon_{fk} = 0.02$, $PgParam = 0.025$ is the optimal value in our experiments.

We saw that the PP is not very effective on composite candidates, so it may be too costly to perform relative to its benefits on most databses (e.g., on Mondial).

4.4.3 Influence of input approximation thresholds on the benefit of the pruning pass

We study in this section how the FK approximation threshold ϵ_{fk} affects the efficiency of the PP. We try our algorithm on Mondial and Janus with several ϵ_{fk}-values (0, 0.01, 0.02), and compare the corresponding shapes of the curve representing the time ratio versus $PgParam$. The curves are displayed in Figure 4.2.

The curves associated with different ϵ_{fk} have the same shape. Smaller values of ϵ_{fk} allow us to process fewer tuples during the pruning pass, so that it is even faster and less costly to discard the accidental candidates. For $\epsilon_{key} = 0.005$, the time ratio associated with the optimal value of the pruning parameter ($PgParam \approx \epsilon_{fk}$) gets smaller when ϵ_{fk} decreases. We cannot compare the curves corresponding to different values of ϵ_{key} for Janus because the number of keys is significantly larger when $\epsilon_{key} = 0.005$ than when $\epsilon_{key} = 0$.

We can see on Mondial that as ϵ_{fk} grows, the left part of the curve is more and more truncated. Then when ϵ_{fk} reaches the value 0.04, the only part that will remain is the part for which the PP is detrimental.

So the FK approximation threshold influences the effectiveness of the pruning pass insofar as it provides a lower bound for the pruning parameter. We saw that the PP is most effective for small values of $PgParam$, therefore when a large ϵ_{fk}-value is

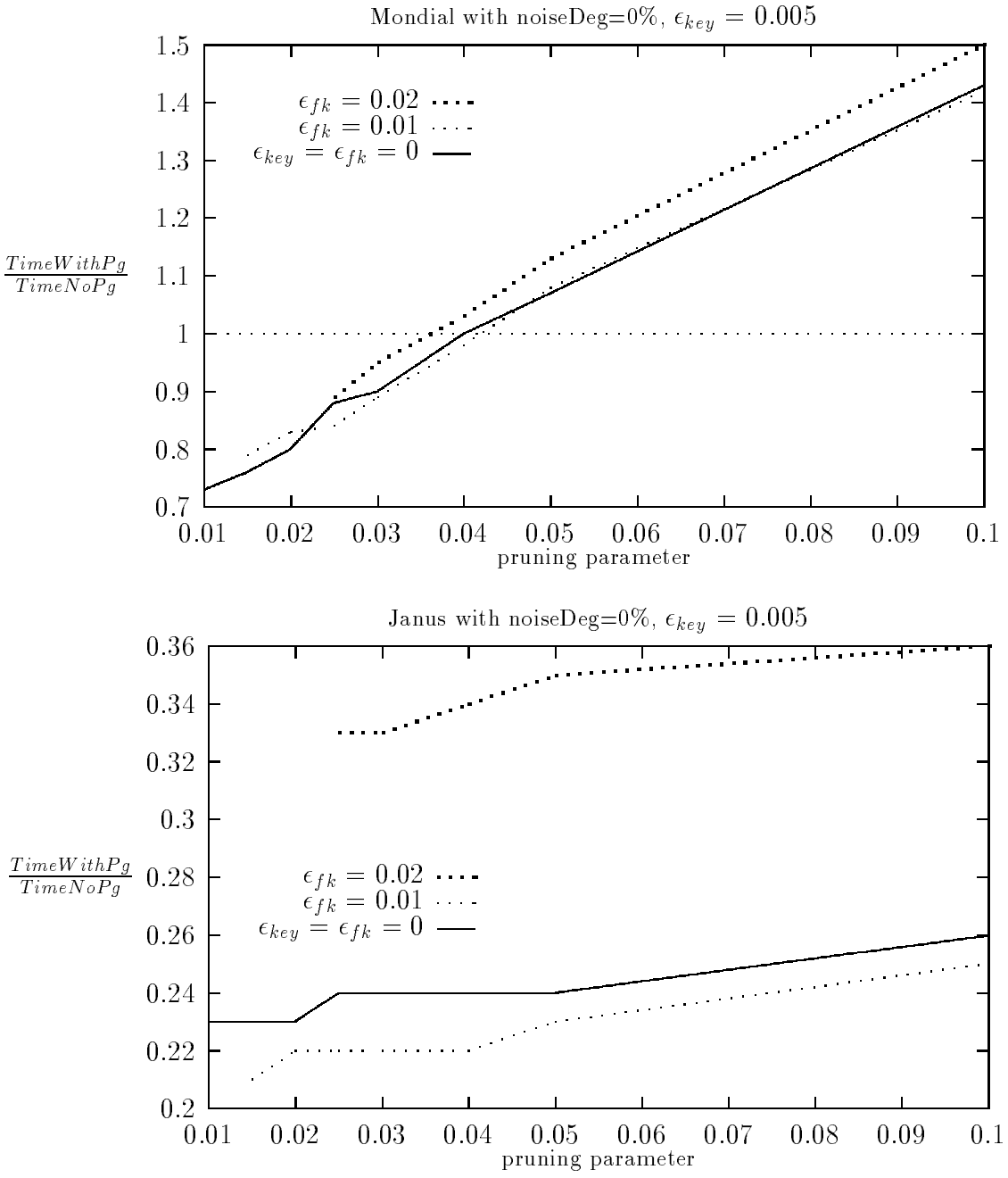

Figure 4.2: Efficiency of the pruning pass for various ϵ_{fk}-values

input to our algorithm, it may be more effective not to perform the PP.

4.4.4 Influence of data size

We found in Section 3.4 that the cost of the pruning pass is in $O(PgParam.p^2)$, where p is the maximal number of tuples per table in the database. We now attempt to verify experimentally that the pruning pass is indeed quadratic w.r.t. the data size.

We chose one of the synthetic datasets for this experiment, because it was easy to increase the size of the data without altering the characteristics of the dependencies. We draw in Figure 4.3 the running time of the algorithm versus the number of tuples in a table.

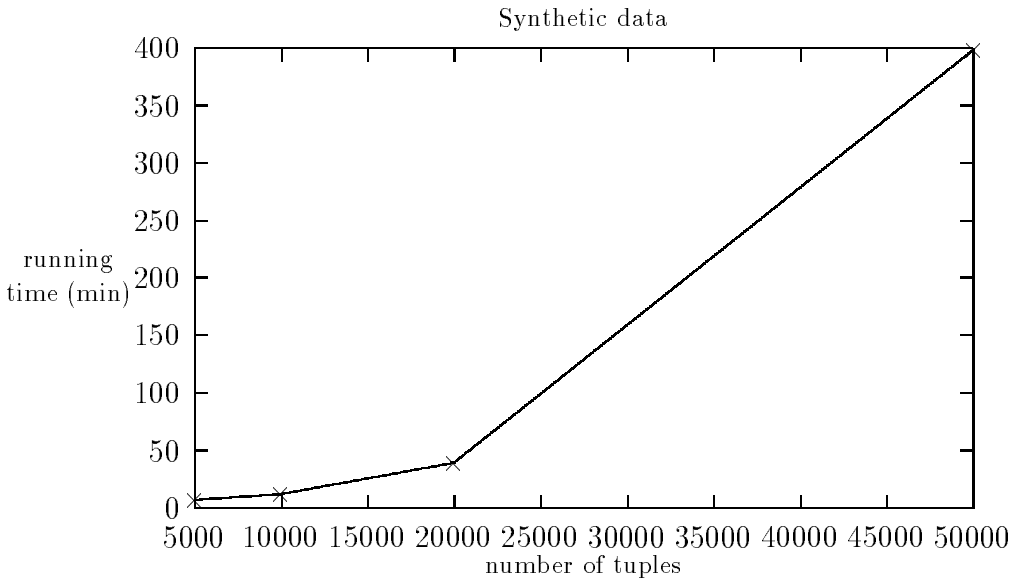

Figure 4.3: Running time on synthetic data, with $\epsilon_{key} = 0.005, \epsilon_{fk} = 0.01, PgParam = 0.015$

The graph in Figure 4.3 is consistent with the shape of a polynomial of degree 2.

Chapter 5

Conclusion

In this thesis, we studied the problem of discovering (approximate) keys and foreign keys holding in a relational database, of which we assume no previous knowledge.

The key discovery part of our topic benefits from a large body of work on functional dependencies: efficient algorithms (e.g., Tane [HKPT98]) have been designed for extracting FDs from a database. On the other hand, identifying foreign keys (or more generally inclusion dependencies) is a relatively new data mining area.

We proposed an integrated architecture for determining the (approximate) keys and foreign keys holding in a database. To the best of our knowledge, this is the first concrete proposal of an algorithm for this the problem, even though the underlying ideas are not new. Our algorithm is designed to work with the data stored in a DBMS, but it could be easily adapted to be memory resident.

We presented two optimizations to this integrated architecture:

- extracting only key-based unary inclusion dependencies instead of all unary INDs, as is usually done;

- adding a pruning pass to the foreign key discovery. The goal of this extra step, performed on a small data sample, is to reduce the number of foreign key candidates tested against the database.

The pruning pass (PP) is useful for discarding accidental candidates, i.e., candidates that share very few values with their right-hand side. Indeed, processing a few tuples is enough to make the decision to prune the candidate, so the PP is very effective at low cost on this type of candidate. For instance, the pruning pass is worthwhile when extracting key-based UINDs, because the candidate generating process is not very constraining and creates many accidental candidates. However, when the candidates share more than a few values with their right-hand side, the cost of pruning is too high in regard to the savings in database queries it generates, thus the PP is disadvantageous. This happens for composite FKs, or when the attribute domains overlap too much.

Two input parameters of our algorithm have a major influence on the effectiveness of the pruning pass: the approximation threshold ϵ_{fk} for foreign keys, which controls the degree of approximation allowed in the FKs discovered, and the pruning parameter $PgParam$, which defines the fraction of tuples involved in the pruning pass. These two parameters are constrained by the inequality $PgParam > \epsilon_{fk}$, which must be satisfied for the PP to function. The PP is most effective for small $PgParam$-values, and becomes unfavorable when $PgParam$ grows, so when ϵ_{fk} is too large, the PP can be detrimental.

In summary, the pruning parameter should be set to a value slightly greater than ϵ_{fk}, and the pruning should occur only on unary candidates. Besides, if the attribute domains overlap too much, or if ϵ_{fk} is large, it is often faster to avoid the PP altogether.

The work presented in this thesis can be extended in several ways. One could investigate heuristic optimizations for the pruning pass. A possible improvement would be to dynamically decide whether a pruning pass should be used for a given foreign key candidate, depending on its domain and the domain of the key. When the domains are of type string, because there are so many possible values, it is very unlikely to find the same strings in both the key and the FK candidate, unless the candidate is really a FK. When the domains are of type number on the other hand, it would not be surprising to find the same values in both key and FK candidate even in cases where they are not linked at all.

Bibliography

[Arm74] William Armstrong. Dependency structures of database relationships. In *Proc. of the IFIP Congress*, pages 580–583, Geneva, Switzerland, 1974.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, (VLDB)*, pages 487–499. Morgan Kaufmann, 1994.

[BB95] Siegfried Bell and Peter Brockhausen. Discovery of constraints and data dependencies in databases (extended abstract). In *European Conference on Machine Learning*, volume 914 of *Lecture Notes in Artificial Intelligence*, pages 267–270. Springer Verlag, 1995. Full version appeared as technical report ftp://ftp-ai.informatik.uni-dortmund.de/pub/Reports/report14.ps.Z.

[Bel95] Siegfried Bell. Discovery and maintenance of functional dependencies by independencies. In *Knowledge Discovery and Data Mining*, pages 27–32, 1995.

[BEST98] Francois Bry, Norbert Eisinger, Heribert Schutz, and Sunna Torge. SIC: Satisfiability checking for integrity constraints. In *DDLP*, pages 25–36, 1998.

[BMKL02] Denilson Barbosa, Alberto O. Mendelzon, John Keenleyside, and Kelly A. Lyons. Toxgene: An extensible template-based data generator for xml. In *SIGMOD Conference*, 2002.

[BMT89] Dina Bitton, Jeffrey Millman, and Solveig Torgersen. A feasibility and performance study of dependency inference. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 635–641. IEEE Computer Society, 1989.

[Bou00] Jean-Francois Boulicaut. A KDD framework for database audit. *International Journal of Information Technology and Management*, 1(3):195–207, 2000. Special issue on Information Technologies in Support of Business Processes.

[CFP84] Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *Journal of Computer and System Sciences*, 28(1):29–59, February 1984.

[CGK+99] Qi Cheng, Jarek Gryz, Fred Koo, T. Y. Cliff Leung, Linqi Liu, Xiaoyan Qian, and Berni Schiefer. Implementation of two semantic query optimization techniques in DB2 universal database. In *The VLDB Journal*, pages 687–698, 1999.

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[Cod74] E. F. Codd. Recent investigations in relational database systems. In *Proceedings of the IFIP Congress*, 1974.

[Dat90] *An Introduction to Database Systems.* Addison-Wesley, 5 edition, 1990.

[dJS99] Lurdes Pedro de Jesus and Pedro Sousa. Selection of reverse engineering methods for relational databases. In *3rd International Conference on Software Maintenace and Reengineering*, pages 194–197, Amsterdam, 1999.

[FS99] Peter A. Flach and Iztok Savnik. Database dependency discovery: A machine learning approach. *AI Communications*, 12(3):139–160, 1999.

[Gen] Genex project. http://www.ncgr.org/genex/index.html.

[GGXZ01] Parke Godfrey, Jarek Gryz, Haoping Xu, and Calisto Zuzarte. Exploiting constraint-like data characterizations in query optimization. In *SIGMOD Record*, 2001.

[Hai91] Jean-Luc Hainaut. Database reverse engineering. In *Proceedings of the 10th Conf. on ER Approach*, San Mateo (CA), 1991.

[HKPT98] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 392–401. IEEE Computer Society, 1998.

[KA96] A. J. Knobbe and P. W. Adriaans. Discovering foreign key relations in relational databases. In *The Thirteenth European Meeting on Cybernetics and Systems Research*, volume II of *Cybernetics and Systems*, pages 961–966, 1996.

[KM92] Jyrki Kivinen and Heikki Mannila. Approximate dependency inference from relations. In *4th International Conference on Database Theory* -

ICDT'92, volume 646 of *Lecture Notes in Computer Science*, pages 86–98. Springer, 1992.

[KMRS92] M. Kantola, H. Mannila, K.-J. Raiha, and H. Siirtola. Discovering functional and inclusion dependencies in relational databases. *Journal of Intelligent Systems*, 7:591–607, 1992.

[Lab] F. Laburthe. Choco, an object oriented constraint propagation kernel over finite domains. http://www.choco-constraints.net/index.html.

[LL99] M. Levene and G. Loizou. *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag, 1999.

[LL01] Mark Levene and George Loizou. Guaranteeing no interaction between functional dependencies and tree-like inclusion dependencies. *Theoretical Computer Science*, 254(1-2):683–690, 2001.

[LPL00] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and Armstrong relations. In *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology*, volume 1777 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2000.

[LPT99] S. Lopes, J.-M. Petit, and F. Toumani. Discovery of interesting data dependencies from a workload of SQL statements. In *Principles of Data Mining and Knowledge Discovery, Third European Conference*, volume 1704 of *Lecture Notes in Computer Science*. Springer, 1999.

[LPT02] Stéphane Lopes, Jean-Marc Petit, and Farouk Toumani. Discovering interesting inclusion dependencies: application to logical database tuning. *Information Systems*, 27(1):1–19, 2002.

[LV00] Mark Levene and Millist W. Vincent. Justification for inclusion dependency normal form. *Knowledge and Data Engineering*, 12(2):281–291, 2000.

[May99] Wolfgang May. Information extraction and integration with FLORID: The MONDIAL case study. Technical Report 131, Universität Freiburg, Institut für Informatik, 1999. Available from http://www.informatik.uni-freiburg.de/~may/Mondial/.

[MFH+01] Renée J. Miller, Daniel Fisla, Mary Huang, David Kalmuk, Fei Ku, and Vivian Lee. The Amalgam schema and data integration test suite. http://www.cs.toronto.edu/~miller/amalgam, 2001.

[MHH00] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema mapping as query discovery. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB)*, pages 77–88. Morgan Kaufmann, 2000.

[MLP02] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. Efficient algorithms for mining inclusion dependencies. In *8th International Conference on Extending Database Technology (EDBT 2002)*, volume 2287 of *Lecture Notes in Computer Science*, pages 464–476. Springer, 2002.

[MR87] H. Mannila and K.-J. Raiha. Dependency inference. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases (VLDB'87)*, pages 155–158. Morgan Kaufmann, 1987.

[MR92] H. Mannila and K. Raiha. *The Design of Relational Databases.* Addison-Wesley, 1992.

[MT97] Heikki Mannila and Hannu Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.

[MTV94] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In *AAAI Workshop on Knowledge Discovery in Databases (KDD-94)*, pages 181–192, Seattle, Washington, 1994.

[NC01] Noel Novelli and Rosine Cicchetti. FUN: An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of 8th International Conference on Database Theory - ICDT 2001*, volume 1973 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2001.

[Oce] Ocean Drilling Program. Janus database. http://www-odp.tamu.edu/database.

[PT99] Jean-Marc Petit and Farouk Toumani. Discovering inclusion and approximate dependencies in relational databases. In *Proc. 15mes Journes Bases de Donnes Avances*, pages 323–339, 1999.

[PTBK96] Jean-Marc Petit, Farouk Toumani, Jean-Francois Boulicaut, and Jacques Kouloumdjian. Towards the reverse engineering of denormalized relational databases. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 218–227. IEEE Computer Society, 1996.

[SZWA99] WM. Shen, W. Zhang, X. Wang, and Y. Arens. Model construction with key identification. In *Proceedings of SPIE Symposium on Data Mining and Knowledge Discovery: Theory, Tools, and Technology*, volume 3695 of *SPIE*, 1999.